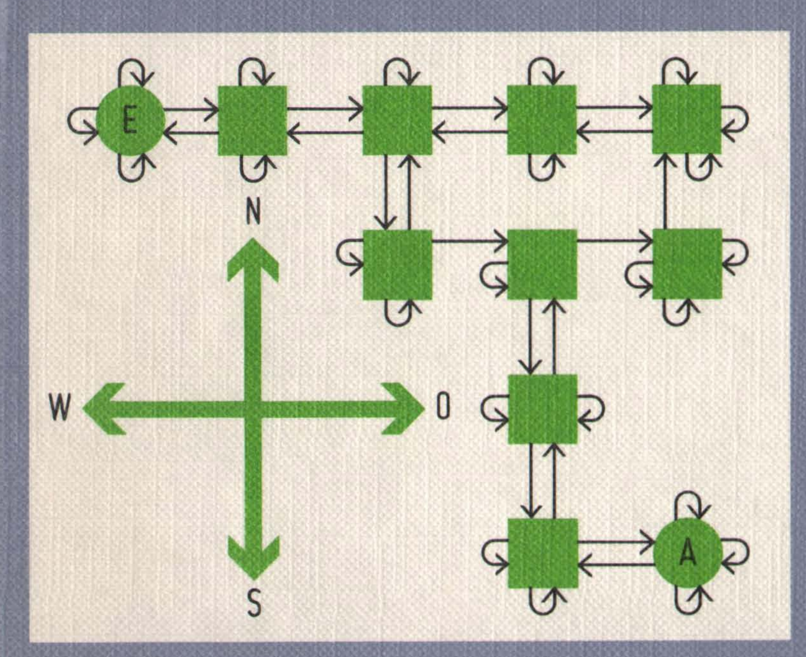
Franzis
Computer-Praxis

Klein

Was ist Pascal?

Eine einfache und kompakte Darstellung
der Programmiersprache mit vielen Beispielen



Programmiersprache PASCAL
PASCAL Grundbefehle
Der Programmaufbau
Die IF-Anweisung
Die REPEAT-Anweisung
PROCEDURE-Deklaration
PASCAL und variable Typen
Gepackte Felder
Zeichenketten
Dynamische Datenstrukturen
Dateiverarbeitung
Sonderfunktionen
Das UCSD-PASCAL
IO-Verarbeitung
PASCAL/M
Konsol-Verarbeitung
PASCAL/Z
Skalare Daten-Typen
PASCAL/MT
Maschinenprogramme
Debugger
Syntaxdiagramme

Franzis'

Klein, Was ist Pascal?

In der Gruppe
Franzis Computer-Praxis
sind erschienen:

Klein, Basic-Interpreter
Klein, Mikrocomputer Hard- und Softwarepraxis
Klein, Mikrocomputersysteme
Klein, Z-80 Applikationsbuch
Plate/Wittstock, Pascal: Einführung - Programmentwicklung - Strukturen

Franzis Computer-Praxis

Rolf-Dieter Klein

Was ist Pascal?

Eine einfache und kompakte Darstellung der Programmiersprache mit vielen Beispielen

Mit 72 Abbildungen



CIP-Kurztitelaufnahme der Deutschen Bibliothek

Klein, Rolf-Dieter:
Was ist Pascal? : Eine einfache u. kompakte Darst. d. Programmiersprache mit vielen Beispielen / Rolf-Dieter Klein. – München : Franzis, 1982.
(Franzis-Computer-Praxis)
ISBN 3-7723-7001-2

Satz: Druckerei Bommer GmbH, Miesbach
Druck: Franzis-Druck GmbH, Karlstraße 35, 8000 München 2
Printed in Germany · Imprimé en Allemagne

ISBN 3-7723-7001-2

Vorwort

PASCAL ist eine höhere Programmiersprache, die Anfang der 70er Jahre von Nikolaus Wirth an der ETH-Zürich entwickelt wurde. Es ist eine strukturierte Sprache, die ähnlich wie Algol 60 ist, aber wesentlich leistungsfähiger.

Das vorliegende Buch richtet sich an den Anfänger, der noch nie programmiert hat, wie auch an Fortgeschrittene, die schon Programmiererfahrung haben. Dabei ist insbesondere an solche Leser gedacht, die vielleicht schon BASIC programmiert haben und nun auf PASCAL umlernen möchten. Das Buch ist auch für Leser geeignet, die sich mit Mikroprozessoren beschäftigen wollen, oder schon damit beschäftigt haben und es sind deshalb in gesonderten Kapiteln Pascal-Systeme beschrieben, die auf Mikrorechnern laufen. Die Beispiele im Buch wurden ebenfalls auf einem Mikrorechner erstellt. Auch die Leser, die sich vorher nur mit Hardware beschäftigt haben und sich nun auch der Software widmen möchten, kommen auf ihre Kosten.

PASCAL dringt immer mehr als Programmiersprache für Mikrorechner vor und es ist daher unerläßlich, sich mit PASCAL zu beschäftigen, wenn man für die Zukunft gerüstet sein und mit Mikrorechnern umgehen will. PASCAL setzt sich aber auch an Großrechnern durch und verdrängt FORTRAN immer mehr, das gerade für technische Aufgaben immer noch sehr oft verwendet wird.

Wichtiger Hinweis

Die in diesem Buch wiedergegebenen Schaltungen und Verfahren werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden*).

Alle Schaltungen und technischen Angaben in diesem Buch wurden vom Autor mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. Der Verlag sieht sich deshalb gezwungen, darauf hinzuweisen, daß er weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernehmen kann. Für die Mitteilung eventueller Fehler sind Autor und Verlag jederzeit dankbar.

---

*) Bei gewerblicher Nutzung ist vorher die Genehmigung des möglichen Lizenzinhabers einzuholen.

Inhalt

1 PASCAL

1.1 Die Programmiersprache PASCAL

1.1.1 Die Entwicklung zu höheren Programmiersprachen

Sehr ähnlich wie bei den Großrechnern verlief auch die Entwicklung bei den Mikroprozessoren. Am Anfang wurde direkt in Maschinensprache programmiert. Dann kamen einfache Assembler hinzu, die zunächst nur auf großen Rechenanlagen lauffähig waren. Sie wurden Cross-Assembler genannt und mit ihnen wird auch heute noch programmiert. Etwas später kamen auch residente Assembler hinzu, die auf dem Mikroprozessorsystem selbst laufen konnten. Mit den Assemblersprachen lassen sich recht leistungsfähige Programme erstellen. Danach wurden einfachere höhere Programmiersprachen wie BASIC oder PL/M (ein Abkömmling von PL/1) und schließlich auch FORTRAN auf Mikrorechnern verfügbar. Als Krönung wurde dann auch PASCAL implementiert. Diese blockorientierte Sprache bietet viele Vorteile, die in den nachfolgenden Kapiteln noch verdeutlicht werden. PASCAL gibt es inzwischen für eine Vielzahl von Mikrorechnern und es verbreitet sich immer mehr.

1.2 PASCAL – Grundbefehle

1.2.1 Grundlagen

Als erstes wollen wir mit einem kleinen Programmbeispiel starten. Es hat die Aufgabe 23 · 14 zu berechnen und auszugeben. *Abb. 1.2.1-1* zeigt das dazugehörige Programm.

```
PROGRAM beispiel(OUTPUT);

  BEGIN
    WRITELN(23*14)
  END.
```

Abb. 1.2.1-1 Einfaches PASCAL-Programm

Es beginnt mit dem Wort PROGRAM, das dem PASCAL-Übersetzer zeigt, daß damit ein PASCAL-Programm beginnt. Danach folgt der Name des Programms, das hier „beispiel" genannt wird. Alle Befehle, die der Compiler versteht, sind in den Programmlistings mit großen Buchstaben geschrieben, um dem Leser zu verdeutlichen, welche Worte frei gewählt werden können und welche eine feste Bedeutung haben und sonst nicht verwendet werden dürfen. Das Wort „beispiel" ist eine solche freie Bezeichnung. In Klammern folgt der Begriff „OUTPUT," der angibt, daß bei dem Programm eine Ausgabe erfolgen soll. Der eigentliche Programmteil ist mit „BEGIN" und „END." geklammert. Die Berechnung und Ausgabe von 23 · 14 erfolgt mit dem Befehl WRITELN (23 · 14).

Die Abfolge von Befehlen kann in PASCAL am elegantesten durch sogenannte Syntaxdiagramme dargestellt werden. *Abb. 1.2.1-2* zeigt die dafür verwendeten Symbole. Beim Beschreibungsaufruf ist in einem Kasten ein Name eingetragen, für den es ein weiteres Syntaxdiagramm gibt. Ein runder Kasten deutet an, daß ein sogenanntes Terminalsymbol vorliegt. Dies sind Zeichen, die der Compiler direkt erkennt

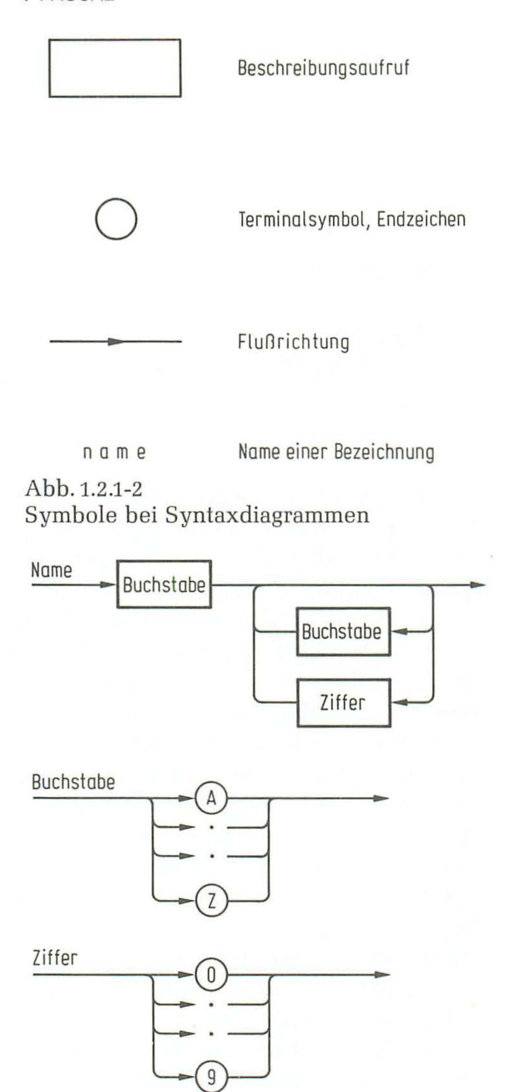


Abb. 1.2.1-2
Symbole bei Syntaxdiagrammen

Abb. 1.2.1-3 Syntaxdiagramm
für einen PASCAL-Namen

und die an dieser Stelle stehen müssen. Mit der Flußrichtung wird angegeben, in welcher Richtung das Syntaxdiagramm zu lesen ist, ähnlich wie bei einem Flußdiagramm. Mit „n a m e" wird schließlich ein Syntaxdiagramm bezeichnet.

Im Anhang sind alle Syntaxdiagramme für die Definition der Sprache PASCAL abgedruckt. Damit kann genau festgestellt werden, in welchen Kombinationen Sprachkonstruktionen von der Definition her erlaubt sind. Das Syntaxdiagramm kann als Nachschlagewerk verwendet werden, um in Zweifelsfällen festzustellen, ob ein entworfener Programmteil vom Compiler akzeptiert werden kann oder nicht.

Abb. 1.2.1-3 zeigt ein kleines Beispiel dafür, das gleich eine praktische Bedeutung besitzt. Freie Bezeichnungen in PASCAL sind z. B. Variablennamen, Prozedurnamen usw. Wie ein solcher Name aufgebaut sein darf, zeigt das Syntaxdiagramm. Dazu werden erst die Begriffe Buchstabe und Ziffer erklärt. Ein Buchstabe kann ein Zeichen „A" bis „Z" sein, eine Ziffer ist ein Zeichen „0" bis „9". Dabei sind „A", oder „0" Terminalsymbole. Ein „Name" kann nun ein Buchstabe sein, ein Buchstabe gefolgt von einem Buchstaben, ein Buchstabe gefolgt von einer Ziffer, ein Buchstabe gefolgt von einem Buchstaben, gefolgt von einem Buchstaben und so weiter. Das heißt, ein Name muß mit einem Buchstaben anfangen, und eine Folge von Buchstaben und Ziffern kann den Rest des Namens bilden.

Beispiel:

Gültige Namen:
X
Y
N14DALPHA
DIESISTEINLANGERNAME

Ungültige Namen:
1C
TEST.NAME
NAMEMIT–

Reservierte Wörter dürfen ebenfalls nicht als Name verwendet werden. Dies geht aber aus dem Syntaxdiagramm nicht hervor, da das Syntaxdiagramm eben nur die Syntax, nicht aber die Semantik beschreibt.

Reservierte Namen sind:

AND	ARRAY	BEGIN
CASE	CONST	DIV
DO	DOWNTO	ELSE
END	FILE	FOR
FUNCTION	GOTO	IF
IN	LABEL	MOD
NIL	NOT	OF
OR	PACKED	PROCEDURE
PROGRAM	RECORD	REPEAT
SET	THEN	TO
TYPE	UNTIL	VAR
WHILE	WITH	

Die Namensliste muß natürlich um die jeweiligen Symbole ergänzt werden, die als Erweiterung in dem Compiler des jeweiligen Herstellers aufgenommen wurden.

Trennzeichen:
Neben den Namen gibt es auch noch weitere Trennzeichen in PASCAL. Ein Beispiel ist das Zeichen „{" oder die Sequenz „(*", die den Anfang eines Kommentars darstellt. Das Ende des Kommentars wird durch das Zeichen „}" oder der Sequenz „*)" angezeigt. Nun kann ein beliebiger Text innerhalb der Kommentarklammern stehen. { Dies ist ein Kommentar in einem PASCAL-Programm } Weitere Trennzeichen sind:

+	−	*	/	:=	.	,
;	:	'	=	⟨⟩	⟨	⟨=
⟩=	⟩	(	)	[	]{	}
↑	..					

Ablauf einer Programmerstellung
Abb. 1.2.1-4 zeigt den Ablaufplan für eine Programmerstellung. Ein PASCAL-Programm wird üblicherweise zunächst mit einem Texteditor eingetippt. Ein Texteditor ist ein Programm, das die Eingabe von Texten ermöglicht. Dabei wird das Programm zunächst auf einem Speichermedium festgehalten, um später dem PASCAL-Compiler zur Verfügung zu stehen. Bei Großrechnern wird als

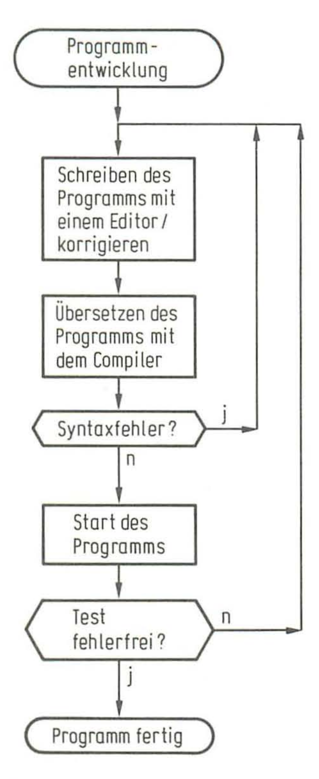


Abb. 1.2.1-4
Ablauf einer Programmentwicklung

Speichermedium meist eine Platte eingesetzt, bei Mikrorechnern kommt üblicherweise eine Floppy dafür in Frage. Das Programm wird anschließend mit dem Compiler übersetzt. Der Compiler stellt dabei eine der Maschine verständlichere Form des PASCAL-Programms her. Gleichzeitig wird aber auch eine Fehlerliste ausgegeben, die angibt, welche Fehler bei der sprachlichen Konstruktion des Programms gemacht wurden: Zum Beispiel die Angabe von PRGRAM, anstelle von PROGRAM bei der Überschrift.

Das Programm muß im Fehlerfall wieder editiert und ein neuer Übersetzerlauf durchgeführt werden.

Ist das Programm fehlerfrei, so kann es gestartet werden. Dabei wird es mit ei-

nem Lader oder durch einen Interpreter in den Hauptspeicher zum Ablauf gebracht. Dabei können wieder Fehler auftauchen, die entweder das Laufzeitsystem entdeckt, wie Fehler DIVISION 1/0 oder die man selbst bemerkt. Es wiederholt sich der Editier- und Übersetzungslauf solange, bis auch der letzte Test als fehlerfrei gilt.

1.2.2 Daten und Zuweisungen

Namen sind ja schon im vorherigen Kapitel erklärt worden und werden hier nur noch angewendet.

Andere wichtige Elemente sind die Darstellung von Zahlen und Zeichen in PASCAL.

Es wird dabei zwischen Ganzen- und Real-Zahlen unterschieden. Ganze-Zahlen zum Beispiel sind: 12 –13 45465600 0. REAL-Zahlen sind: 2.3 –4.5E–3 0.000002.

Die genaue Definition ist im Anhang unter „Ganze Zahl vorzeichenlos" und „Zahl vorzeichenlos" als Syntaxdiagramm dargestellt.

Zeichen werden in Anführungszeichen geschrieben, also z. B.:
'Z' 'Zeichenkette mit Leerzeichen' ' '
Namen können in PASCAL mit einem konstanten Wert versehen werden. Sie heißen dann Konstante.

Damit der PASCAL-Compiler weiß, daß es sich um eine Konstantendefinition handelt, muß das Wort CONST geschrieben werden.
Beispiel:

```
CONST
   pi = 3.1412;
   leerzeichen = ' ';
   maximum = 1000;
```

Die einzelnen Definitionen werden durch Semikolons getrennt. Der Wert dieser Konstanten kann nicht erneut verändert werden. Namen können aber auch als Variable deklariert werden, so daß der Wert beliebig geändert werden kann.

Dazu gibt es das reservierte Wort VAR. Nun genügt aber nicht allein die Angabe eines Namens, sondern es muß auch noch gesagt werden, für welchen Typ der Name verwendet werden soll, also ob er nur Ganze-Zahlen erhalten soll, oder Zeichen usw. Dazu gibt es weitere Symbole, sie lauten:
INTEGER REAL BOOLEAN CHAR

Nun kann mit der Definition begonnen werden. Beispiel:

```
VAR
   zaehler : INTEGER;
   zeichen : CHAR;
   wahr : BOOLEAN;
   ergebnis : REAL;
```

Auch hier wird die Trennung durch Semikolons vorgenommen. Gleiche Typen können aber auch durch Kommatas getrennt werden:

```
VAR
   i, k, zaehler : INTEGER;
   zeichen1, zeichen2 : CHAR;
```

Mit diesen Namen kann nun gearbeitet werden. Der Name „zeichen1" soll z. B. für eine spätere Ausgabe der Wert '.' bekommen, dies wird durch eine Zuweisung erreicht und sieht dann so aus:

```
zeichen1 : = '.'
```

„zähler" soll den Wert 5 erhalten:

```
zaehler : = 5
```

Nun kann nicht nur zugewiesen, sondern auch gerechnet werden.

```
zaehler : = zaehler + 1;
ergebnis : = 23 · 45 + 3
```

Für mathematische Operationen gibt es Standardfunktionen und Operatoren:

+ – · /	vier Grundrechenarten
( )	für Verschachtelungen
DIV MOD	Ganzzahldivision und Modulofunktion
ABS (X)	Absolutbetrag
SQR (X)	Das Quadrat einer Zahl

TRUNC (×)	Abschneiden der Nachkommastellen
ROUND (×)	Runden einer Zahl
SIN (×)	Sinuswert
COS (×)	Cosinus
ARCTAN (×)	Arcustangens
LN (×)	Natürlicher Logarithmus
EXP (×)	Exponentialfunktion
SQRT (×)	Quadratwurzel einer Zahl

Dabei ist zu beachten, daß, falls ganzzahlige Argumente verwendet werden, auch reale Zahlen entstehen können.
Beispiel:

1 / 3	hat den Wert 0.333333
20 MOD 3	besitzt den Wert 2

Dabei können anstelle der Zahlen auch Variablen vom Typ INTEGER oder REAL stehen, die den entsprechenden Wert haben.

Das Ergebnis ist im ersten Fall vom Typ REAL und im zweiten vom Typ INTEGER. Eine Zuweisung eines INTEGER-Wertes an einen REAL-Typ kann erfolgen, aber nicht umgekehrt.
Bisher wurde der Typ BOOLEAN noch nicht besprochen.

Eine Boolesche Variable kann den Wert TRUE oder FALSE besitzen. Für boolesche Variablen gibt es auch eine Reihe von erlaubten Operationen wie:

AND OR NOT

Beispiel:

```
  CONST
     wahr = TRUE;
  VAR
     vielleicht, luege,
     aussage : BOOLEAN;
```

Dann ist folgendes möglich:

```
     luege : = NOT wahr;
     vielleicht : = aussage AND wahr
     AND NOT luege;
```

Wobei hier „vielleicht“ den Wert von „aussage“ annimmt, da „wahr AND NOT luege“ wieder TRUE ist.

Ferner gibt es noch Operationen – auch Vergleiche genannt – die als Argumente Zahlen besitzen und einen Booleschen Wert als Ergebnis liefern:

= <> <= >= < > IN

Die Argumente können auch Boolesche Werte sein:

1 < 2	ergibt den Wert TRUE.
0 = 1	ergibt den Wert FALSE.
TRUE <> FALSE ist	TRUE

IN wird bei dem Kapitel über Mengen erklärt.

Es gibt aber auch noch andere Funktionen:

ODD (×)	ergibt TRUE wenn × ungerade ist.
EOLN (×)	ergibt TRUE wenn ein Zeilenende beim Einlesen erkannt wurde.
EOF (×)	TRUE wenn Dateiende vorliegt.

Für den Typ CHAR gibt es auch Funktionen.

ORD (c)	ergibt die Ordnungszahl des Zeichens. Welcher Wert dabei entsteht, hängt sehr stark vom Rechner ab. Bei Mikrorechnern wird üblicherweise der ASCII-Wert verwendet. Das Ergebnis ist ein INTEGER-Wert.
CHR (×)	Ist die Umkehrfunktion zu ORD. CHR (ORD(c)) ergibt c.

Die Relationen sind anhand der Relation ihrer Ordnungszahlen erklärt. Es gilt also dann:

'A' < 'B'	falls ORD('A') < ORD('B') TRUE ist.

Das Gleiche gilt auch für die anderen Relationen.

Zwei weitere Funktionen sind interessant:

PRED (c)	ergibt den Vorgänger des Zeichens c also CHR(ORD(c)-1)
SUCC (c)	entspricht CHR(ORD(c)+1)

1.2.3 Der Programmaufbau

Bei PASCAL ist ein festes Gerüst für den Programmaufbau vorgeschrieben. Jedes PASCAL-Programm beginnt mit der Überschrift „PROGRAM". Dann folgt ein Deklarationsteil, indem zuerst alle Konstanten definiert werden, dann alle Typen (siehe Kapitel über Typen), dann alle Variable und schließlich Funktionen und Prozeduren. Erst danach folgt nach „BEGIN" und „END." geklammert, das eigentliche Programm. Die Struktur sieht also folgendermaßen aus:

```
PROGRAM
   Deklarationen
   BEGIN
      Anweisungen
   END.
```

PASCAL benötigt ferner in der Überschrift noch einen Programmnamen und die Angabe von verwendeten Ein- und Ausgabegeräten.

Dazu ein Beispiel. In *Abb. 1.2.3-1* ist ein kurzes Programm dargestellt. Der Name des Programms lautet „beispiel". Die Angaben von „INPUT" und „OUTPUT" bedeuten, daß hier eine Eingabe und eine Ausgabe von Daten, z. B. auf einer Konsole erfolgen soll. Die Überschrift wird durch ein Semikolon vom Rest getrennt. Dann folgt die Definition einer Variablen „a". Da hier keine Konstanten gebraucht werden, wird der entsprechende Deklarationsteil mit „CONST" weggelassen. Durch „a : REAL" wird a als Variable für reelle Zahlen definiert.

Das Programm startet dann mit der Anweisung „READ(a)" die besagt, daß der Wert der Variablen a eingelesen werden soll. Anschließend wird mit dem Befehl „WRITELN(...)" die Ausgabe eines Textes und einer Berechnung veranlaßt.

Nach Starten des übersetzten Programms, wartet dieses zunächst auf die Eingabe einer REAL-Größe. Hier wird die Zahl 12 eingegeben. Damit der Rechner weiß, wann die Eingabe der Zahl abgeschlossen ist, muß ein beliebiges an-

```
PROGRAM beispiel(INPUT,OUTPUT);

  VAR
     a : REAL;

  BEGIN
     READ(a);
     WRITELN('13% MWST von ',a,' DM sind: ',a * 0.13)
  END.

12

13% MWST von  1.20000e1 DM sind:  1.56000e0
```

Abb. 1.2.3-1 Mehrwertsteuer Berechnung

deres Zeichen außer Punkt, Zahl oder Vorzeichen und des Buchstabens „E" eingegeben werden. Üblicherweise wird ein CR (carriage return oder Wagenrücklauf) oder ein Leerzeichen eingegeben.

Die Ausgabe von a erfolgt in REAL, ebenfalls das Ergebnis der Berechnung a · 0.13. Soll eine andere Darstellung als die mathematische erreicht werden, so muß ein entsprechendes Programmteil dafür geschrieben werden.

Mit READ können die Datentypen INTEGER, REAL und CHAR eingelesen und mit WRITELN ausgegeben werden. Dabei gibt es noch die Form READLN, wobei, falls die Eingabe aller Variablen erledigt ist, bis zum Zeilenende (CR) alle Zeichen ignoriert werden. Bei WRITE wird im Gegensatz zu WRITELN nach der Ausgabe der Werte kein CR LF (Wagenrücklauf und Zeilenvorschub) ausgegeben.

Die Parameter werden durch Kommatas getrennt (siehe auch Syntaxdiagramme im Anhang).

1.2.4 Die IF-Anweisung

Nach den bisherigen Kenntnissen ist es nun möglich, einfache lineare Programme zu entwerfen. Die einzelnen Anweisungen werden hintereinander geschrieben und nur durch Semikolons getrennt. Beispiel:

```
PROGRAM linear;
  BEGIN
    anweisung1;
    anweisung2;
    anweisung3
  END.
```

Nun, damit allein kann nicht vernünftig programmiert werden. Daher gibt es in PASCAL, wie auch in anderen Programmiersprachen Anweisungen, die den Ablauf eines Programmes in Abhängigkeit von Bedingungen steuern können.

Mit der IF-Anweisung kann eine solche Fallunterscheidung durchgeführt werden.

Das Format für die IF-Anweisung sieht dann so aus:

```
IF bedingung THEN anweisung
```

oder:

```
IF bedingung THEN anweisung1
ELSE anweisung2
```

Es gibt also zwei verschiedene Formen. Bei der ersten wird, falls die Bedingung den Wert TRUE hat, die nach THEN stehende Anweisung ausgeführt und falls die Bedingung den Wert FALSE hat, wird sie nicht ausgeführt. In beiden Fällen wird eine ggf. danach stehende Anweisung ausgeführt.

Bei der zweiten Form wird, falls die Bedingung den Wert TRUE hat, „anweisung1" ausgeführt und falls der Wert FALSE ist wird „anweisung2" ausgeführt.

Abb. 1.2.4-1 zeigt ein Beispiel für die erste Form. Es soll die Quadratwurzel einer Zahl berechnet werden. Dabei ergibt sich ein imaginärer Wert, wenn das Argument unter der Wurzel negativ ist. Wird die Operation SQRT(×) für ein negatives × ausgeführt, so gibt es eine Fehlermeldung bei der Ausführung. Um die Berechnung dennoch ausführen zu können, muß eine Fallunterscheidung getroffen werden. Im ersten Fall ist das Argument größer oder gleich Null, im zweiten ist es kleiner als Null.

Im Programmbeispiel nach der ersten Variante der IF-Anweisung, wird nach der Eingabeanweisung der Variablen „zahl" abgefragt, ob der Wert von „zahl" kleiner als Null ist. Wenn ja, so wird die Ausgabe des Textes „I ·" veranlaßt um anzuzeigen, daß das Ergebnis Komplex ist. In jedem Fall wird der Wert der Qua-

```
PROGRAM wurzel(INPUT,OUTPUT);
   VAR
      zahl:INTEGER;
   BEGIN
      WRITELN;
      WRITE('Eingabe einer Zahl:');
      READ(zahl);
      WRITE('Wurzel aus ',zahl,' ist ');
      IF zahl<0 THEN WRITE(' I *');
      WRITELN(SQRT(ABS(zahl)))
   END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
Eingabe einer Zahl:13
Wurzel aus 13 ist  3.60555e0

Sorcim PASCAL/M ver 03.05/03.05
Eingabe einer Zahl:-4
Wurzel aus -4 ist  I * 2.00000e0
```

Abb. 1.2.4-1
Wurzel einer Zahl
mit IF THEN

```
PROGRAM wurzel(INPUT,OUTPUT);
   VAR
      zahl : INTEGER;
   BEGIN
      WRITELN;
      WRITE('Eingabe einer Zahl:');
      READ(zahl);
      WRITE(' Wurzel aus ',zahl,' ist ');
      IF zahl < 0 THEN WRITE(' I *',SQRT(-zahl))
                  ELSE WRITE(SQRT(zahl))
   END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
Eingabe einer Zahl:34
 Wurzel aus 34 ist  5.83095e0

Sorcim PASCAL/M ver 03.05/03.05
Eingabe einer Zahl:-12
 Wurzel aus -12 ist  I * 3.46410e0
```

Abb. 1.2.4-2
Wurzel einer Zahl
mit IF THEN ELSE

dratwurzel durch die Anweisung WRITELN(SQRT(ABS(zahl)) ausgegeben. Durch die Operation ABS(zahl) wird erreicht, daß das Argument der Quadratwurzel in jedem Fall positiv ist.

Abb. 1.2.4-2 zeigt eine Lösung, mit der zweiten Variante, eine IF-Anweisung aufzubauen. Ist der Wert von „zahl" kleiner Null, so wird direkt mit SQRT(-zahl) der richtige Wert ausgegeben, ist der Wert von „zahl" nicht negativ, so wird im ELSE-Teil mit SQRT(zahl) der Wert berechnet.

Nun taucht die Frage auf, wie können mehrere Anweisungen im THEN oder ELSE Teil angegeben werden?

Dies ist ganz einfach, wenn man sich einmal die Definition einer Anweisung ansieht.

```
Anweisung : diverse Anweisungen
          : BEGIN Anweisungen
                  durch Semiko-
                  lon getrennt
                                END
```

Eine Anweisung kann einmal sein: „diverse Anweisungen", wie z. B. die IF-Anweisung, oder eine Zuweisung, oder READ, WRITE Befehle usw.

Oder eine Anweisung kann aus Anweisungen bestehen, die durch Semikolons getrennt sind und als Ganzes mit BEGIN und END geklammert sind.

Beispiel:

```
IF bedingung
  THEN
    BEGIN
      anweisung1;
      anweisung2;
    END
  ELSE
    anweisung3
```

In den Anweisungen „anweisung1 ... anweisung3" dürfen natürlich wieder IF-Anweisungen verwendet werden. Die sind durch den rekursiven Charakter der Programmiersprache möglich und auch im Syntaxdiagramm (Anhang) ersichtlich.

Als Bedingung darf ein beliebiger Ausdruck mit einem Booleschen Wert stehen, also auch Boolesche Variablen im einfachsten Fall.

1.2.5 Die REPEAT-Anweisung

Mit der IF-Anweisung lassen sich nun schon Entscheidungen treffen. Dies genügt aber nicht für alle Programmaufgaben. Soll ein Vorgang mehrfach wiederholt werden, so wird eine Schleife benötigt. In BASIC läßt sich eine solche Schleife mit einem Sprungbefehl verwirklichen, dies ist aber nicht immer sehr durchsichtig; in PASCAL gibt es bessere Möglichkeiten.

Die REPEAT Anweisung ist eine davon. Das Format lautet wie folgt:

```
REPEAT
  anweisungen
UNTIL bedingung
```

Die zwischen REPEAT und UNTIL stehende(n) Anweisung(en) wird (werden) so lange ausgeführt, bis der Ausdruck „bedingung" den Wert TRUE annimmt, also die Bedingung erfüllt ist.

Beispiel und Vergleich zu BASIC:

```
REPEAT
  READ(a);
  verarbeite a
UNTIL fertig
```

In BASIC:

```
100 INPUT A
110 verarbeite a
120 IF NOT(fertig) THEN 100
```

„Fertig" kann dabei eine Boolesche Variable sein, oder ein Ausdruck, der einen Booleschen Wert besitzt, z. B. „a=0".

Abb. 1.2.5-1 zeigt ein einfaches Programmbeispiel dazu. Es soll die Mehrwertsteuer berechnet werden. Diesmal

```
PROGRAM beispiel(INPUT,OUTPUT);

   VAR
      wert : REAL;

   BEGIN
      WRITELN;
      WRITELN('Berechnen der Mehrwertsteuer');
      REPEAT
         WRITE(' Eingabe DM Betrag ');
         READ(wert);
         WRITELN(' 13% MWST von ',wert,' sind ',wert*0.13)
      UNTIL wert = 0
   END.
```

```
Berechnen der Mehrwertsteuer
 Eingabe DM Betrag 12
 13% MWST von  1.20000e1 sind  1.56000e0
 Eingabe DM Betrag 40.01
 13% MWST von  4.00100e1 sind  5.20130e0
 Eingabe DM Betrag 10.45
 13% MWST von  1.04500e1 sind  1.35850e0
 Eingabe DM Betrag 0
 13% MWST von  0.00000e0 sind  0.00000e0
```

Abb. 1.2.5-1
Mehrwertsteuer mit
REPEAT-Anweisung

soll aber nicht für jede neue Berechnung das Programm neu aufgerufen werden, sondern die Berechnung soll solange ausgeführt werden, bis durch ein Endezeichen dem Programm gezeigt wird, daß die Berechnung abgeschlossen ist. Als Endekriterium wird hier einfach der DM Betrag 0 genommen, da er in dem normalen Berechnungsvorgang nicht vorkommt. Das Programm beginnt nach der Definition der Variablen „wert" mit der Ausgabe der Überschrift. Dann wird in der Schleife die Eingabe des DM-Betrags verlangt und nach dem Einlesen des Wertes wird die berechnete Mehrwertsteuer ausgegeben. Hat die Variable „wert" den Wert 0 angenommen, so wird die Schleife beendet. Ein kleiner Nachteil bei diesem Verfahren ist, daß auch mit dem Wert 0 die Berechnung durchgeführt wird. Dieser Schönheitsfehler läßt sich jedoch z. B. durch eine IF-Anweisung vermeiden, oder durch Programmkonstruktionen, die im weiteren noch erklärt werden.

Abb. 1.2.5-2 zeigt ein weiteres Beispiel zur REPEAT-Anweisung. Es soll eine Rechnung erstellt werden. Dazu wird jeweils die Anzahl der Ware eingegeben und der Wert der Ware. Am Schluß erwarten wir die Rechnungssumme.

Für diese Aufgabe benötigt man mehrere Variablen. In „wert" wird der jeweilige Warenwert eingegeben. „summe"

```
PROGRAM rechnung(INPUT,OUTPUT);

   VAR
      wert,summe : REAL;
      anzahl : INTEGER;

   BEGIN
      WRITELN;
      WRITELN(' Rechnung ');
      summe := 0;
      READ(anzahl,wert);
      REPEAT
         WRITELN(anzahl,' * ',wert,'          = ',anzahl*wert);
         summe := summe + anzahl * wert;
         READ(anzahl,wert)
      UNTIL anzahl = 0;
      WRITELN;
      WRITELN('---------------------------------');
      WRITELN('    Gesamt: ',summe )
   END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
 Rechnung
3,5.10
3 *  5.10000e0          =  1.53000e1
7,1
7 *  1.00000e0          =  7.00000e0
10,22
10 *  2.20000e1          =  2.20000e2
0,0

---------------------------------
   Gesamt:  2.42300e2
```

Abb. 1.2.5-2 Rechnungsstellung mit REPEAT

speichert die Endsumme der Rechnung und in „anzahl" wird die Anzahl der Ware eingegeben. Nach Ausgabe der Überschrift, muß als erstes die Variable „summe" mit 0 belegt werden, damit sie einen definierten Startwert hat. Es wird dann mit READ(anzahl,summe) die Anzahl der Waren und deren Preis eingelesen. Dies geschieht vor dem Start der Schleife um hier den Fehler des vorherigen Programms zu vermeiden, um am Schluß einen nicht bedeutenden Wert zu verarbeiten. In der Schleife wird dann als erstes die Berechnung „anzahl · wert" durchgeführt und eine Zeile der Rechnung ausgegeben. Der Wert von „summe" wird um den Betrag „anzahl · wert" erhöht. Dann wird wieder „anzahl" und „summe" eingelesen. Ist als Anzahl der Wert 0 eingegeben worden, so wird die Schleife abgebrochen und die Gesamtsumme, die in der Variablen „summe" steht, ausgegeben.

Abb. 1.2.5-3 zeigt eine andere Verwendungsmöglichkeit der REPEAT-Anweisung. Es soll ein Plot einer Funktion, einer Sinuskurve, ausgegeben werden. Dazu wird hier eine weitere Fähigkeit von WRITE und WRITELN verwendet. es ist dabei möglich, die Anzahl der ver-

```
PROGRAM sinus(OUTPUT);
   CONST
      pi = 3.14159;

   VAR
      lauf : REAL;
      position : INTEGER;

   BEGIN
      WRITELN;
      lauf := 0;
      REPEAT
         position := 40 + TRUNC(SIN(lauf)*40);
         WRITELN(1:position);
         lauf := lauf + pi/12;
      UNTIL lauf > 2*pi
   END.
```

Abb. 1.2.5-3 Plotten von Funktionen mit REPEAT

```
                                       1
                                                 1
                                                          1
                                                                    1
                                                                          1
                                                                              1
                                                                                1
                                                                              1
                                                                          1
                                                                    1
                                                            1
                                                 1
                                       1
                             1
                    1
           1
     1
 1
1
 1
     1
           1
                   1
                              1
```

wendeten Stellen zur Darstellung einer Zahl anzugeben. WRITE(wert:stellenzahl) gibt die Variable „wert" mit „stellenzahl" Stellen aus. Nun kann „stellenzahl" auch eine Variable sein und damit ist es möglich, die Anweisung zum Plotten zu mißbrauchen. In dem Programm wird als erstes pi als Konstante definiert und mit dem entsprechenden Wert belegt. „lauf" ist eine REAL-Größe, die als Schleifenzähler verwendet wird. Es wird dann als erstes „lauf" mit 0 belegt und der Anfang festgelegt. Die Variable „position" soll die Position des Plotpunktes erhaltenund wird dazu mit dem Wert des Ausdrucks: 40+TRUNC(SIN(lauf) · 40) belegt. SIN besitzt einen Wertebereich von −1 bis +1. Durch die Multiplikation mit dem Wert 40 wird der Bereich auf −40 bis +40 ausgedehnt. Mit TRUNC wird die entstandene Zahl, die immer noch eine REAL-Größe ist, in einen INTEGER-Wert verwandelt, dadurch, daß die Nachkommastellen abgeschnitten werden. Nun wird noch 40 aufaddiert, und es ergibt sich ein Be-

reich von 0 bis 80. Mit dem Ausgabebefehl wird die Position als Stellenzahl interpretiert und die vorangestellte Zahl mit der entsprechenden Gesamtstellenzahl ausgeben, wobei die nicht verwendeten Positionselemente als Leerzeichen ausgegeben werden.

Auch dieses Programm enthält einen Schönheitsfehler. Nimmt die Position den Wert 0 an, so müßte die Zahl 1 mit 0 Stellen dargestellt werden. Da dies nicht möglich ist, ergibt sich hier ein Fehler bei der Darstellung. Der Fehler läßt sich leicht durch eine Addition von 1 beheben. Die Ausgabe müßte dann wie folgt lauten: WRITELN(1:position+1);

oder es wird bei der Berechnung von „position“ nicht 40 sondern 41 addiert.

Ferner ist natürlich bei der Ausgabe darauf zu achten, daß das ausgegebene Bild auf die Bildfläche in horizontaler Richtung paßt. Eine Anpassung kann aber leicht durch ändern der Werte in der Berechnungsformel von „position“ durchgeführt werden.

1.2.6 Die WHILE-Anweisung

In PASCAL gibt es auch noch eine weitere ähnliche Schleifenkonstruktion wie die REPEAT-Anweisung. Bei der REPEAT-Anweisung erfolgt die Abfrage einer Bedingung am Schluß. Bei WHILE ist dies anders. Das Format sieht wie folgt aus:

```
PROGRAM beispiel(INPUT,OUTPUT);
   VAR
      zahl,count:INTEGER;
   BEGIN
      WRITELN;
      WRITE('Eingabe von Teiler:');
      READ(zahl);
      count:=1;
      WRITE('Teiler von ',zahl,' sind ');
      WHILE count<=zahl DO
         BEGIN
            IF zahl MOD count = 0 THEN WRITE(count,' ');
            count := count + 1
         END
   END.

Sorcim PASCAL/M ver 03.05/03.05
Eingabe von Teiler:234
Teiler von 234 sind 1 2 3 6 9 13 18 26 39 78 117 234

Sorcim PASCAL/M ver 03.05/03.05
Eingabe von Teiler:512
Teiler von 512 sind 1 2 4 8 16 32 64 128 256 512
```

Abb. 1.2.6-1 Teilerberechnung mit WHILE

```
PROGRAM wandle(INPUT,OUTPUT);

   VAR
      zahl,temp : INTEGER;

   BEGIN
      WRITELN;
      WRITELN(' Dezimal nach Binaer ');
      REPEAT
         READ(zahl);
         temp := zahl;
         WRITELN(' von LSB nach MSB ');
         WHILE temp > 0 DO
            BEGIN
               IF TRUNC(temp/2)*2 = temp
                  THEN
                     WRITELN('0')
                  ELSE
                     WRITELN('1');
               temp := TRUNC(temp / 2)
            END;
         WRITELN;
      UNTIL zahl = 0
   END.
```

Abb. 1.2.6-2
Binärausgabe
mit WHILE

```
 Dezimal nach Binaer
12
 von LSB nach MSB
0
0
1
1

20000
 von LSB nach MSB
0
0
0
0
0
1
0
0
0
1
```

```
1
1
0
0
1

255
 von LSB nach MSB
1
1
1
1
1
1
1
1

0
 von LSB nach MSB
```

WHILE bedingung DO anweisung
Solange der Ausdruck „bedingung" den Wert TRUE besitzt, wird die nach DO folgende Anweisung wiederholt ausgeführt. Sollen mehrere Anweisungen ausgeführt werden, so werden diese mit BEGIN und END geklammert, gemäß der Definition von „anweisung" (Syntaxdiagramm). Da hier zuerst die Bedingung abgefragt und dann die Anweisung ausgeführt wird, ergeben sich ganz andere Konstruktionsmöglichkeiten als bei der REPEAT-Anweisung. Ist die Bedingung zum Beispiel von Anfang an FALSE, so wird „anweisung" überhaupt nicht ausgeführt. Bei der REPEAT-Anweisung jedoch muß die Anweisung in der Schleife mindestens einmal durchlaufen werden.

Abb. 1.2.6-1 zeigt ein Beispiel mit der WHILE-Anweisung. Aufgabe ist es, alle Teiler einer Zahl auszugeben. Die Variable „zahl" beinhaltet die Zahl, von der die Teiler berechnet werden sollen. „count" ist ein Zähler, der von 1 bis „zahl" läuft. Dazu wird „zahl" zunächst mit dem Wert 1 belegt. Dann erfolgt der Eintritt in die Schleife. Die Bedingung „count<=zahl" ist beim ersten Mal immer dann erfüllt, wenn eine zahl größer 0 eingegeben wurde. In der Schleife wird geprüft, ob „zahl MOD count = 0" ist. Wenn ja, so ist „count" ein Teiler von „zahl". „count" wird dann um eins erhöht

Abb. 1.2.6-2 zeigt nun ein kombiniertes Beispiel. Eine eingegebene Zahl soll in das duale Zahlensystem umgewandelt werden. Wird die Zahl 0 eingegeben, so ist dies das Ende der Eingabe und die Ausführung soll abgebrochen werden. Nach Ausgabe der Überschrift begint die eigentliche Hauptschleife, die mit „REPEAT, UNTIL zahl = 0" geschachtelt ist. In dieser Schleife wird der Wert von „zahl" eingelesen. Dann wird eine Unterüberschrift ausgegeben und es startet eine innere Schleife, die mit WHILE ausgeführt ist. In dieser Schleife wird solange geblieben, bis die Variable „temp" einen Wert kleiner oder gleich 0 hat. „temp" wird dabei mit dem Wert von „zahl" vorbelegt. Nun folgt in der Schleife, die durch BEGIN und END geklammert ist, die Ausgabe der einzelnen Binärstellen. Dazu wird ein einfacher Algorithmus verwendet. Ist „temp" durch 2 teilbar, so wird die Zahl „0" ausgegeben, sonst die Zahl 1. Dann wird „temp" durch den ganzzahligen Teil der Division von „temp" durch zwei ersetzt. Ist „temp" gleich 0, so kann der Algorithmus abgebrochen werden. Die Stellen der Zahl werden beginnend mit der niederwertigsten Stelle ausgegeben.

Abb. 1.2.6-3 zeigt die Lösung einer ganz anderen Aufgabe. Sie stammt aus dem Bereich der Stringverarbeitung. (Zeichenketten-Verarbeitung).

Abb. 1.2.6-4 zeigt das Syntaxdiagramm der Aufgabenstellung. Es soll eine Zahl eingelesen werden, bei der beliebig viele Plus-Zeichen vorne stehen dürfen, und die von beliebig vielen Pluszeichen gefolgt werden darf. Tritt ein anderes Zeichen auf, so soll der Vorgang beendet werden und eine Ausgabe erfolgen.

Die Variable „ch" wird vom Typ CHAR erklärt und kann damit ein Zeichen aufnehmen. Es wird nach Programmstart ein Zeichen von der Konsole gelesen. Mit der WHILE-Schleife wird nun solange gelesen, wie das eingelesene Zeichen ein Plus-Zeichen ist. Wenn schon das erste Zeichen kein Plus-Zeichen war, wird die Anweisung wegen der WHILE-Konstruktion übersprungen und für ch steht eine Ziffer. Nun muß aus einer Folge von Ziffern eine Zahl gewonnen werden. Dazu wird die INTEGER-Variable

```
PROGRAM leszeich(INPUT,OUTPUT);
  VAR
    ch:CHAR;
    zahl:INTEGER;
  BEGIN
    WRITELN;
    READ(ch);
    WHILE ch='+' DO READ(ch);
    zahl:=0;
    REPEAT
      zahl := (ORD(ch)-ORD('0'))+10*zahl;
      READ(ch)
    UNTIL ch='+';
    READ(ch);
    WHILE ch='+' DO READ(ch);
    WRITELN('Zahl ist',zahl)
  END.
```

Abb. 1.2.6-3
Zeichenerkennung mit WHILE

```
Sorcim PASCAL/M ver 03.05/03.05
+++++345+++++ Zahl ist345
```

```
Sorcim PASCAL/M ver 03.05/03.05
+++23++++++++++ Zahl ist23
```

„zahl" zunächst mit dem Wert 0 belegt. Dann beginnt die innere Schleife. Es wird der Wert der Zahl berechnet. Dazu wird die Funktion ORD verwendet. Um rechnerunabhängig zu bleiben, wird eine Differenz von zwei mit ORD gebildeten Werten erzeugt. Der alte Wert der Zahl wird mit 10 multipliziert und dann aufaddiert, so daß eine Eingabe beginnend mit der höherwertigen Stelle möglich ist. Nach dieser Schleife erfolgt mit einer weiteren WHILE-Schleife das Ignorieren von nachfolgenden Plus-Zeichen. Eine Fehlerbehandlung wurde hier noch nicht vorgesehen: also wenn zum Beispiel in der REPEAT-Schleife ein anderes Zeichen außer „+" oder einer Ziffer eingegeben wird. Elegante Lösungen dazu lernen wir noch später kennen.

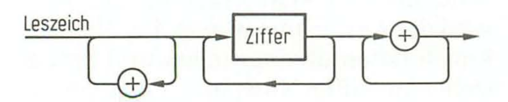


Abb. 1.2.6-4
Syntaxdiagramm für Eingabesprache

1.2.7 Die CASE-Anweisung

In *Abb. 1.2.7-1* ist ein Programm gezeigt, das die Aufgabe hat, eine eingegebene Ziffer im Klartext auszugeben; beispielsweise die Ziffer 0 wird als „Null"

```
PROGRAM numaus(INPUT,OUTPUT);

  VAR
    zahl : INTEGER;

  BEGIN
    REPEAT
      WRITELN;
      WRITE('Eingabe einer Zahl:');
      READ(zahl);
      IF zahl = 0 THEN WRITELN('Null');
      IF zahl = 1 THEN WRITELN('Eins');
      IF zahl = 2 THEN WRITELN('Zwei');
      IF zahl = 3 THEN WRITELN('Drei');
      IF zahl = 4 THEN WRITELN('Vier');
      IF zahl = 5 THEN WRITELN('Fuenf');
      IF zahl = 6 THEN WRITELN('Sechs');
      IF zahl = 7 THEN WRITELN('Sieben');
      IF zahl = 8 THEN WRITELN('acht');
      IF zahl = 9 THEN WRITELN('neun');
      WRITELN
    UNTIL zahl = 10;
  END.
```

Abb. 1.2.7-1
CASE durch IF-Anweisungen

```
Eingabe einer Zahl:0
Null


Eingabe einer Zahl:1
Eins


Eingabe einer Zahl:9
neun


Eingabe einer Zahl:10
```

ausgegeben. Wird die Zahl 10 eingegeben, so wird das Programm beendet. Nach der Eingabe der Variablen „zahl" wird hier einfach durch mehrere IF-Anweisungen die jeweilige Zahl abgefragt und eine entsprechende Ausgabe ausgeführt. Dies ist sehr umständlich. Es gibt in PASCAL eine elegantere Lösung mit der CASE-Anweisung.

Das Format der CASE-Anweisung lautet wie folgt:

```
CASE ausdruck OF
   case marke : anweisung 1 ;
   . . .
   case marke : anweisung n
END
```

Der hinter dem Wort CASE angegebene Ausdruck bestimmt, welches der nachfolgenden Anweisungen ausgeführt wird. Es wird dann die Ausführung auf die entsprechende „case marke" über-

```
PROGRAM numaus(INPUT,OUTPUT);

   VAR
      zahl : INTEGER;

   BEGIN
      REPEAT
         WRITELN;
         WRITE('Eingabe einer Zahl:');
         READ(zahl);
         IF zahl < 10
            THEN
               CASE zahl OF
                  0: WRITELN('Null');
                  1: WRITELN('Eins');
                  2: WRITELN('Zwei');
                  3: WRITELN('Drei');
                  4: WRITELN('Vier');
                  5: WRITELN('Fuenf');
                  6: WRITELN('Sechs');
                  7: WRITELN('Sieben');
                  8: WRITELN('Acht');
                  9: WRITELN('Neun')
               END;
         WRITELN
      UNTIL zahl = 10
   END.
```

Abb. 1.2.7-2
Die Anwendung der CASE-Anweisung

```
Eingabe einer Zahl:0
Null


Eingabe einer Zahl:5
Fuenf


Eingabe einer Zahl:1
Eins


Eingabe einer Zahl:9
Neun


Eingabe einer Zahl:10
```

tragen. Als Ausdruck sind alle skalaren oder Bereichstypen zugelassen, wenn nur die „case marke“ vom gleichen Typ ist. Wird im Ausdruck ein Wert errechnet, der keine entsprechende Marke besitzt, so weiß man nicht, was dann im Rechner passiert. Bei der CASE-Anweisung können auch mehrere „case-marken“ für eine Anweisung angegeben werden, sie werden dazu durch Kommatas getrennt:

case marke1,
case marke2 : anweisung

Abb. 1.2.7-2 zeigt die Lösung der vorherigen Aufgabe mit der CASE-Anweisung. Da bei Eingabe der Zahl 10 das Programm beendet werden soll, und die Zahl 10 nicht als „case-marke“ vorkommt, muß der Wert durch eine IF-Anweisung abgefangen werden.

Mit den bisherigen Kenntnissen ist es nun möglich, auch eine komplizierte Aufgabe zu lösen. Es soll eine Art EDITOR programmiert werden, mit dem einfach PASCAL-Programme eingegeben werden können. Sobald der EDITOR ein PASCAL-Wort an seinen Anfangsbuchstaben erkannt hat, soll er den Rest des Wortes ausgeben. Um zu unterscheiden, ob es ein PASCAL-Wort sein kann, oder eine Variable, muß hier die Abmachung eingehalten werden, alle PASCAL-Schlüsselwörter mit Großbuchstaben zu schreiben. Eine Zeichenfolge die von „'“ umklammert ist, soll immer direkt ausgegeben werden, da dort Groß- und Kleinbuchstaben akzeptiert werden müssen, um z. B. bei WRITELN beliebige Textausgabe zu erlauben. Das Programm wird hier nur auf der Konsole ausgegeben, da uns bis jetzt noch die Datenverwaltung fehlt, doch ist dies nur eine leichte Änderung. Das Programm verwendet nur eine Variable vom Typ CHAR, mit dem Namen „ch“. Dann beginnt die Hauptschleife, die mit REPEAT , UNTIL aufgebaut ist und endet, wenn in der Variablen „ch“ das Zeichen „.“ steht. Es wird ein Zeichen eingelesen. Handelte es sich um das Zeichen „'“, so wird solange eingelesen, bis wieder ein Anführungszeichen erscheint. Dann kommt eine Abfrage, ob ein Zeichen ein Großbuchstabe war. Falls ja, so wird eine CASE-Anweisung ausgeführt. Es wird nun auf die Schlüsselwörter hin abgefragt. Dabei werden nicht alle Schlüsselwörter von PASCAL geprüft, um das Programm nicht unnötig kompliziert zu machen. Folgende Wörter werden erkannt:

VAR	INTEGER	INPUT
CHAR	OUTPUT	BEGIN
END	WHILE	WRITE
WRITELN	DO	REPEAT
READ	READLN	LN(
UNTIL		

Abb. 1.2.7-3 zeigt die Programmausführung. Dabei ist LN(die Endung von „WRITELN(“ und „READLN(“. Die CASE-Anweisung prüft nun die Anfangsbuchstaben. Bei manchen Schlüsselwörtern reicht dies aus, aber natürlich nicht bei allen. Dazu wird innerhalb einer Anweisung hinter der „case marke“ eine weitere CASE-Anweisung ausgeführt. Bei manchen Schlüsselwörtern wird auch gleich ein Semikolon ausgegeben oder ein Zeilenvorschub. Auf diesem Prinzip könnte ein recht komfortabler EDITOR geschrieben werden, der Tippfehler vermeidet und zu einer schnellen Eingabe des PASCAL-Programms verhilft.

1.2.8 Die FOR-Anweisung

In vielen Fällen wird bei Programmen in einer Zählervariablen ein Wert hochgezählt und eine Anweisungssequenz wiederholt. Diesen Programmiervorgang unterstützt PASCAL mit der FOR-

```
PROGRAM prgein(INPUT,OUTPUT);
   VAR
      ch:CHAR;
   BEGIN
      WRITELN;
      WRITE('PROGRAM ');
      REPEAT
         READ(ch);
         IF ch='''' THEN
            REPEAT
               READ(ch)
            UNTIL ch='''';
         IF (ORD(ch)<=ORD('Z')) and (ORD(ch)>=ORD('A')) THEN
            CASE ch OF
               'V': WRITELN('AR');
               'I': BEGIN
                       READ(ch); (* N *)
                       READ(ch);
                       CASE ch OF
                          'P': WRITE('UT');
                          'T': WRITELN('EGER;')
                       END
                    END;
               'C': WRITELN('HAR;');
               'O': WRITE('UTPUT');
               'B': WRITELN('EGIN');
               'E': WRITE('ND');
               'W': BEGIN
                       READ(ch);
                       CASE ch OF
                          'H': WRITE('ILE ');
                          'R': WRITE('ITE')
                       END
                    END;
               'D': WRITELN('O');
               'R': BEGIN
                       READ(ch);
                       READ(ch); (* Erster RE *)
                       CASE ch OF
                          'P': WRITELN('EAT');
                          'A': WRITE('D')
                       END
                    END;
               'L': WRITE('N(');
               'U': WRITE('NTIL ')
            END
      UNTIL ch='.'
   END.
```

Abb. 1.2.7-3 Programm-Eingabe mit CASE

```
Sorcim PASCAL/M ver 03.05/03.05
PROGRAM test(INPUT,OUTPUT);
   VAR
      i:INTEGER;
   BEGIN
      REPEAT
         READ(i);
         WRITELN(i,'I ausgeben')
      UNTIL i=0
   END.
```

Anweisung. Das Format dazu ist folgendes:

```
FOR variable := ausdruck1
      TO ausdruck2
   DO anweisung
```

oder Alternativ:

```
FOR variable := ausdruck1
      DOWNTO ausdruck2
   DO anweisung
```

Beim ersten Durchlauf wird die Variable mit dem Wert von „ausdruck1" belegt, dann wird im ersten Fall bei jedem Durchlauf der Wert der Variablen durch den Nachfolger (Funktion SUCC( )) ersetzt, bis ausdruck2 überschritten wird. Bei INTEGER-Variablen wird durch SUCC(X) einfach der Wert der Variablen um eins erhöht. REAL-Variablen sind als Zähler nicht zugelassen, da die Funktion SUCC(X) nicht zugelassen ist. Im zweiten Fall, bei der Konstruktion mit DOWNTO, wird die Funktion PRED(X) verwendet und bei einer INTEGER-Variablen der Wert um eins erniedrigt. Der Abbruch erfolgt dann bei Unterschreiten des Wertes von ausdruck2. Zum besseren Verständnis eine gleichwertige Konstruktion mit der WHILE-Anweisung:

Fall 1 FOR mit TO:

```
variable := ausdruck1;
WHILE variable <= ausdruck2 DO
  BEGIN
    anweisung;
    variable := SUCC(variable)
  END
```

Fall 2 mit DOWNTO:

```
variable := ausdruck1;
WHILE variable >= ausdruck2 DO
  BEGIN
    anweisung;
    variable := PRED(variable)
  END
```

```
PROGRAM tabelle(INPUT,OUTPUT);
  VAR
     i:INTEGER;
  BEGIN
     WRITELN;
     FOR i:=0 TO 10 DO
        WRITELN('Wurzel aus ',i,' ist ',SQRT(i))
  END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
Wurzel aus 0 ist  0.00000e0
Wurzel aus 1 ist  1.00000e0
Wurzel aus 2 ist  1.41421e0
Wurzel aus 3 ist  1.73205e0
Wurzel aus 4 ist  2.00000e0
Wurzel aus 5 ist  2.23607e0
Wurzel aus 6 ist  2.44949e0
Wurzel aus 7 ist  2.64575e0
Wurzel aus 8 ist  2.82843e0
Wurzel aus 9 ist  3.00000e0
Wurzel aus 10 ist  3.16228e0
```

Abb. 1.2.8-1 Wurzeltabelle mit FOR-Anweisung

```
PROGRAM tabelle(INPUT,OUTPUT);
   CONST
      pi=3.14121;
   VAR
      i:INTEGER;
      a:REAL;
   BEGIN
      WRITELN;
      a:=0;
      FOR i:=20 DOWNTO 0 DO
         BEGIN
            WRITELN(i,' sin(',a,') ist ',SIN(a));
            a:=a+pi/10
         END
   END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
20 sin( 0.00000e0) ist 0.00000e0
19 sin( 3.14121e-1) ist  3.08981e-1
18 sin( 6.28242e-1) ist  5.87723e-1
17 sin( 9.42363e-1) ist  8.08950e-1
16 sin( 1.25648e0) ist 9.51009e-1
15 sin( 1.57061e0) ist 1.00000e0
14 sin( 1.88473e0) ist 9.51127e-1
13 sin( 2.19885e0) ist 8.09174e-1
12 sin( 2.51297e0) ist 5.88033e-1
11 sin( 2.82709e0) ist 3.09344e-1
10 sin( 3.14121e0) ist 3.82543e-4
9 sin( 3.45533e0) ist -3.08617e-1
8 sin( 3.76945e0) ist -5.87414e-1
7 sin( 4.08357e0) ist -8.08725e-1
6 sin( 4.39769e0) ist -9.50891e-1
5 sin( 4.71182e0) ist -1.00000e0
4 sin( 5.02594e0) ist -9.51245e-1
3 sin( 5.34006e0) ist -8.09398e-1
2 sin( 5.65418e0) ist -5.88341e-1
1 sin( 5.96830e0) ist -3.09707e-1
0 sin( 6.28242e0) ist -7.63178e-4
```

Abb. 1.2.8-2
Sinustabelle mit
FOR-Anweisung

Abb. 1.2.8-1 zeigt ein Beispiel. Es sollen die Wurzeln der zahlen 0 bis 10 ausgegeben werden. In *Abb. 1.2.8-2* ein weiteres Beispiel: diesmal mit der DOWNTO-Version der For-Anweisung. Es wird eine Sinustabelle mit 20 Einträgen ausgegeben. Da als Schleifenzähler keine REAL-Zahl zugelassen ist, muß hier in einer zweiten Variablen das Argument für die Sinuswerte festgehalten werden.

1.3 PASCAL-Prozeduren und Funktionen

1.3.1 PROCEDURE-Deklaration

Wird ein Programmabschnitt mehrere Male benötigt, so ist es recht praktisch, ein Unterprogramm daraus zu machen. In PASCAL gibt es aber auch noch andere Gründe, ein Unterprogramm – PROCEDURE genannt – zu bilden. Prozeduren werden am Anfang eines Blocks definiert. Ein Block sieht wie folgt aus

```
PROGRAM unterprg(INPUT,OUTPUT);

   VAR
      sum,a,b : INTEGER;


   PROCEDURE addier;

      BEGIN
         sum := a + b;
      END;

BEGIN
   WRITELN;
   REPEAT
      READ(a,b);
      addier;
      WRITELN(' summe = ',sum)
   UNTIL sum = 0
END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
1
2
 summe = 3
5
123
 summe = 128
-10
2
 summe = -8
0
0
0
 summe = 0
```

Abb. 1.3.1-1 Unterprogrammtechnik

Block: Konstantendefinitionsteil
Variablendefinitionen
Prozeduredefinitionen
BEGIN durch „;“
getrennte Anweisungen
END

Im Syntaxdiagramm des Anhangs ist die Definition ausführlicher dargestellt. Die Prozedur-Definition lautet:

entweder: PROCEDURE name ; block
oder: PROCEDURE name
parameterliste ; block

parameterliste:
(durch Semikolon getrennt sind
namen, die durch Kommas getrennt sind : Typ)

Da in der Prozedur-Deklaration wieder ein „block“ verwendet werden kann, ist es also auch möglich, dort neue Prozedur Deklarationen einzuführen.

Die Vielfalt läßt sich am besten anhand von Beispielen kennenlernen.

```
PROGRAM unterprg(INPUT,OUTPUT);
   VAR
      zahl:INTEGER;
      ch:CHAR;

   PROCEDURE plusweg;
      BEGIN
         REPEAT
            READ(ch)
         UNTIL ch<>'+'
      END;

   PROCEDURE lesint;
      BEGIN
         zahl:=0;
         REPEAT
            zahl:=zahl*10+(ORD(ch)-ORD('0'));
            READ(ch)
         UNTIL ch='+'
      END;

   BEGIN
      WRITELN;
      plusweg;
      lesint;
      plusweg;
      WRITELN(' zahl ',zahl)
   END.

   Sorcim PASCAL/M ver 03.05/03.05
   +++++123++++  zahl 123
```

Abb. 1.3.1-2
Zeichenerkennung mit Unterprogramm

Abb. 1.3.1-1 zeigt ein kleines Programm, daß die Summe zweier eingegebener Werte ausgeben soll. Dazu ist ein Unterprogramm mit dem Namen „addier“ definiert. Es bildet in der Variablen „sum“ die Summe von „a“ und „b“. Die Variablen sind dabei im Definitionsteil des Hauptprogramms angegeben und vom Typ INTEGER. Diese Variablen sind, wie man sagt, GLOBAL. Sie gelten damit auch in der Definition des Unterprogramms. Im Hauptprogrammteil wird einfach der Name des definierten Unterprogramms angegeben und es wird dann dort aufgerufen.

Abb. 1.3.1-2 zeigt die Lösung der Programmieraufgabe des vorigen Kapitels für die Aufgabe eine Zahl einzulesen und dabei führende und nachfolgende Plus-Zeichen zu ignorieren. Es werden dazu zwei Prozeduren definiert. Die Prozedur „plusweg“ hat die Aufgabe, solange ein Zeichen einzulesen, bis dieses ungleich dem Plus-Zeichen ist. Die Prozedur „lesint“ liest Ziffern ein, wandelt diese in eine Zahl um, bis ein Plus-Zeichen in den Eingabedaten erscheint. Im Hauptprogramm läßt sich damit eine sehr übersichtliche Programmstruktur erreichen.

Prozeduren können aber auch Parameter erhalten. Es werden dabei zwei verschiedene Arten von Parametern verwendet. Ein Parameter, der mit VAR deklariert ist, wird an das Unterprogramm in Form der Adresse übergeben, so daß das Unterprogramm Manipulationen direkt an der Variablen des aufrufenden Programms durchführt. Wird der Parameter nicht mit VAR definiert, so

```
PROGRAM unterprg(INPUT,OUTPUT);

   VAR
      sum,a,b : INTEGER;

   PROCEDURE addier(VAR summe:INTEGER; x,y:INTEGER);

      BEGIN
         summe := x + y
      END;

BEGIN
   WRITELN;
   READ(a,b);
   addier(sum,a,b);
   WRITELN(' summe = ',sum)
END.


Sorcim PASCAL/M ver 03.05/03.05
34
56
 summe = 90
```

Abb. 1.3.1-3 VAR-Parameter

```
PROGRAM uprparameter(INPUT,OUTPUT);

   VAR
      i : INTEGER;

   PROCEDURE upraufruf(i : INTEGER);

      BEGIN
         WRITELN('wert i in UPR ist: ',i);
         i := 5
      END;

   PROCEDURE uprmitvar(var i : INTEGER);

      BEGIN
         WRITELN('wert i in UPRVAR: ',i);
         i := 6
      END;


   BEGIN
      WRITELN;
      i := 12;
      upraufruf(i);
      WRITELN('wert i in hauptprg: ',i);
      i := 10;
      WRITELN;
      uprmitvar(i);
      WRITELN('wert i in hauptprg: ',i);
   END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
wert i in UPR ist: 12
wert i in hauptprg: 12

wert i in UPRVAR: 10
wert i in hauptprg: 6
```

Abb. 1.3.1-4 Wirkung von Parametern

wird der Wert der Variablen an das Unterprogramm übergeben, und dort in eine eigene LOKALE Variable gestellt. *Abb. 1.3.1-3* zeigt ein Beispiel für den Gebrauch von Parametern. Die Variable „summe“ ist dabei mit VAR definiert, da der Wert, der ihr im Unterprogramm zugewiesen wird, auch im Hauptprogramm noch erreichbar sein muß.

Das Beispiel in *Abb. 1.3.1-4* verdeutlicht noch einmal den Unterschied zwischen einem durch VAR deklarierten Parameter und einem normal deklarierten. LOKALE Variable können aber auch innerhalb der Prozedur definiert werden. Sie gelten dann aber auch nur innerhalb dieser Prozedur. *Abb. 1.3.1-5* zeigt dazu ein Beispiel. Werden Varia-

```
PROGRAM unterprg(INPUT,OUTPUT);

   VAR
      a,b,c : INTEGER;

   PROCEDURE wechsel;

      VAR
         b : INTEGER;

      BEGIN
         b:=1;
         c:=2;
         WRITELN('UPR: b=',b,' c=',c)
      END;

BEGIN
   WRITELN;
   a:=3;
   b:=4;
   c:=5;
   WRITELN('HPR: a=',a,' b=',b,' c=',c);
   wechsel;
   WRITELN('HPR: a=',a,' b=',b,' c=',c);
END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
HPR: a=3 b=4 c=5
UPR: b=1 c=2
HPR: a=3 b=4 c=2
```

Abb. 1.3.1-5 Gültigkeitsbereich von Namen

```
PROGRAM uprtechnik(INPUT,OUTPUT);

   VAR
      i : INTEGER;
      zahl : REAL;

   PROCEDURE ausgabe(w : REAL; n : INTEGER);

      VAR
         temp,i : INTEGER;
         rest : REAL;

      BEGIN (* Ausgabe eines REAL Wertes mit max n Nachkommas *)
         temp := TRUNC(w); (* Ganzahliger Anteil *)
         WRITE(temp);
         rest := w - TRUNC(w);
         WRITE('.');
         FOR i := 1 TO n DO
            BEGIN
               temp := TRUNC(rest*10);
               WRITE(temp);
               rest := 10*rest - TRUNC(10*rest)
            END
      END;

BEGIN (* Hauptprogramm *)

   WRITELN;
   REPEAT
      WRITE(' Eingabe zahl in REAL und Nachkommas ');
      READ(zahl,i);
      WRITELN;
      ausgabe(zahl,i);
      WRITELN
   UNTIL zahl=0
END.
```

Abb. 1.3.1-6 Nachkommaausgabe

```
Sorcim PASCAL/M ver 03.05/03.05
 Eingabe zahl in REAL und Nachkommas 23.34 4

23.3400
 Eingabe zahl in REAL und Nachkommas 12 3

12.000
 Eingabe zahl in REAL und Nachkommas 0.23 2

0.22
 Eingabe zahl in REAL und Nachkommas 0.2345 3

0.234
 Eingabe zahl in REAL und Nachkommas 12.34353 1

12.3
 Eingabe zahl in REAL und Nachkommas 0 1

0.0
```

```
PROGRAM uprverschachteln(INPUT,OUTPUT);

   PROCEDURE aussen(t1 : INTEGER);
      VAR
         a1 : INTEGER;

      PROCEDURE innen;
         VAR
            i1 : INTEGER;

         BEGIN
            i1 := 1;
            aussen(1);
            WRITELN('a1 ',a1,' i1 ',i1)
         END;

      BEGIN (* aussen *)
         a1 := 2;
         IF t1=1 THEN a1 := 3;
         IF t1=2 THEN innen;
         WRITELN('a1 ',a1);
         WRITELN('t1 ',t1)
      END;

BEGIN (* Hauptprogramm *)
   WRITELN;
   aussen(2);
   WRITELN
END.

Sorcim PASCAL/M ver 03.05/03.05
a1 3
t1 1
a1 2 i1 1
a1 2
t1 2
```

Abb. 1.3.1-7 Unterprogramm-verschachtelung

blen in einem Unterprogramm mit dem gleichen Namen definiert, wie in einem übergeordneten Programmteil, so ist die Variable des äußeren Teils im Inneren nicht zugänglich. Umgekehrt können Variablen, die im inneren Teil deklariert wurden, von außen nie erreicht werden, egal ob sie einen gleichen oder einen anderen Namen haben.

Abb. 1.3.1-6 zeigt einen praktischen Gebrauch der bisherigen Erfahrungen. Die Prozedur „ausgabe“ kann eine REAL-Zahl mit einer vorgegebenen Zahl von Nachkommastellen ausgeben.

```
PROGRAM hanoi(INPUT,OUTPUT); (* aus Grogono [ 1 ] *)
   VAR
      total : INTEGER;

   PROCEDURE schiebeturm( hoehe,von,nach,mit : INTEGER );

      PROCEDURE schiebplatte( nimmweg,setze : INTEGER );

         BEGIN
            WRITELN(nimmweg, '->',setze)
         END;

      BEGIN (* schiebeturm *)
         IF hoehe > 0
            THEN
               BEGIN
                  schiebeturm(hoehe-1,von,mit,nach);
                  schiebplatte(von,nach);
                  schiebeturm(hoehe-1,mit,nach,von)
               END
      END;

BEGIN (* Hauptptogramm *)
   WRITELN;
   WRITE('Anzahl: ');
   READ(total);
   WRITELN;
   schiebeturm(total,1,3,2)
END.


Sorcim PASCAL/M ver 03.05/03.05
Anzahl: 4

1->2
1->3
2->3
1->2
3->1
3->2
1->2
1->3
2->3
2->1
3->1
2->3
1->2
1->3
2->3
```

Abb. 1.3.1-8 Die Türme von Hanoi

Unterprogramme lassen sich aber auch verschachteln. So können sich zwei Prozeduren z. B. gegenseitig aufrufen, wenn sie wie in *Abb. 1.3.1-7* konstruiert sind. Das Beispiel besitzt hier keine weitere praktische Bedeutung, aber die Aufrufe lassen sich recht gut verfolgen. Der Aufruf von „aussen(2)“ bewirkt den Aufruf der Prozedur „aussen“. Der Parameter 2 wird an die dazu lokale Variable „t1“ übergeben. In der Prozedur „aussen“ ist eine weitere Prozedur „innen“ definiert. Das Programm startet dann mit der Zuweisung der Zahl 2 an die Variable a1. Es folgen dann zwei Abfragen. Ist „t1=1“, so wird an a1 der Wert 3 zugewiesen. Ist t1=2, dann wird die Prozedur „innen“ aufgerufen, was jetzt der Fall ist. In „innen“ wird die Variable „i1“ mit 1 belegt, um zu zeigen, daß der Wert auch bei erneutem Aufruf von aussen erhalten bleibt. „aussen“ wird diesmal mit dem Wert 1 aufgerufen und damit wird dort nicht erneut „innen“ aufgerufen. Danach wird der Wert von „a1“ und „i1“ ausgegeben und wieder nach „aussen“ zurückgekehrt. Dann folgt die Ausgabe von „a1“ und „t1“.

Noch etwas mehr ist ein Beispiel verschachtelt, daß *Abb. 1.3.1-8* zeigt. Es geht um das Problem der Türme von Hannoi. *Abb. 1.3.1-9* zeigt eine schematische Anordnung des Spiels. Es sind drei Türme vorhanden. Auf dem Turm 1 befindet sich eine Anzahl von Scheiben, die nach oben hin mit abnehmenden Durchmesser angeordnet sind. Diese Scheiben müssen nun auf Turm 3 gebracht werden, mit Hilfe von Turm 2. Dabei darf immer nur eine Scheibe von einem Turm auf einen anderen transportiert werden und es darf nie eine größere Scheibe auf einer kleineren liegen.

Unser Programm löst dieses Problem und gibt die Sequenz aus, in der die

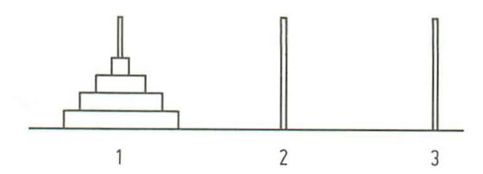


Abb. 1.3.1-9 Spielsituation zu 3.1.3.1-8

Scheiben auf die jeweils angegebenen Türme gelegt werden müssen. Bei größeren Werten für die Anzahl nimmt dabei die Zahl der benötigten Züge rapide zu.

Soll eine Prozedur z. B. in einer anderen Prozedur aufgerufen werden, bei der sie noch nicht definiert ist, so kann dies durch eine FORWARD-Deklaration durchgeführt werden. Die FORWARD-Prozedur soll umgekehrt wieder die erste Prozedur aufrufen. *Abb. 1.3.1-10* zeigt den Gebrauch der FORWARD-Deklaration. Normalerweise läßt sie sich aber dadurch umgehen, daß bei rekursiven Aufrufen die Deklarationen ineinander geschachtelt werden. Die Parameter werden nur im FORWARD-Teil definiert. Die eigentliche Definition des Prozedur-Rumpfes folgt später ohne Angabe von Parametern.

Ein Beispiel wie es auch ohne FORWARD geht, zeigt *Abb. 1.3.1-11*. Aufgabe ist es, durch Eingabe einer Booleschen Formel daraus eine Schaltung zu konstruieren. Vielleicht läßt sich daraus einmal eine Art Hardwarekonstruktionssprache entwickeln. es sind die Verknüpfungen „+“ für ODER, „·“ für UND und „–“ für die NEGATION definiert. Mit Klammern können die Operationen verschachtelt werden. Das Syntaxdiagramm der Eingabesprache ähnelt der Definition einer Expression (Ausdruck), nur daß alle Zahlen-Arithmetiken nicht vorhanden sind. Der Übersetzungsvorgang ähnelt sehr dem eines Compilers für eine Programmiersprache.

```
PROGRAM forwaerts(INPUT,OUTPUT);

   PROCEDURE spaeter(k : INTEGER); FORWARD;

   PROCEDURE vorher(l : INTEGER);

      BEGIN
         WRITELN('vorher ',l);
         IF l=1 THEN spaeter(l-1);
         WRITELN('vorher ende ',l)
      END;

   PROCEDURE spaeter;

      BEGIN
         WRITELN('spaeter ',k);
         IF k=0 THEN vorher(k);
         WRITELN('spaeter ende ',k)
      END;

BEGIN
   WRITELN;
   WRITELN(' verschachtelter aufruf ');
   vorher(1);
   WRITELN(' *********************** ')
END.

B>prun procfor
Sorcim PASCAL/M ver 03.05/03.05
 verschachtelter aufruf
vorher 1
spaeter 0
vorher 0
vorher ende 0
spaeter ende 0
vorher ende 1
 ***********************
```

Abb. 1.3.1-10 FORWARD-Prozeduren

```
 1   program kons1;
 1
 1    (* Konstrukta    RDK 810904 V1.0      *)
 1    (* Hardware Konstruktionssprache     *)
 1
 1   const
 1
 1      maxstr = 9;
 1
 1   type
 1      strtyp = packed array[0..maxstr] of char;
 1
 1
 1   var
 1      nextchar : char; {einlese zeichen }
 1      value : strtyp; {generiertes symbol }
 1      wert : strtyp; {fuer ausgabe }
 1
 1   procedure initnew(var s : strtyp);
 1
 1     var
 1       i : integer;
 1
 1     begin
 1       for i:=0 to maxstr do s[i] := ' ';
 3       s[0] := '&';
 4       s[1] := '0'
 5     end;
 5
 5
 5
 5   procedure gennew(var s : strtyp); {generiere next symbol }
 5
 5      var
 5        s1 : strtyp;
 5        i : integer;
 5
 5      begin
 5        s1 := s;
 6        for i := 0 to maxstr do s[i] := ' ';
 8        s[0] := '&';
 9        s[1] := succ(s1[1])
10      end;
10
10   procedure readchar;
10
10      begin
10        read(nextchar)
11      end;
11
11
11
11   procedure undaus(var e1,e2,a : strtyp);
11
11     begin
11       writeln('            ','    ---    ');
```

Abb. 1.3.1-11
Hardware-Compiler
(PASCAL/Z) als Beispiel

```
12        writeln(      e1       ,'---| |    ');
13        writeln('              ','    |&|---',a);
14        writeln(      e2       ,'---| |    ');
15        writeln('              ','    ---    ')
16      end;
16
16    procedure oderaus(var e1,e2,a : strtyp);
16
16      begin
16        writeln('              ','    ---    ');
17        writeln(      e1       ,'---|>|    ');
18        writeln('              ','    |=|---',a);
19        writeln(      e2       ,'---|1|    ');
20        writeln('              ','    ---    ')
21      end;
21
21    procedure nichtaus(var e,a : strtyp);
21
21      begin
21        writeln('              ','    ---    ');
22        writeln(     e         ,'---| |O---',a);
23        writeln('              ','    ---    ')
24      end;
24
24
24
24    procedure expr(var exprval : strtyp);
24
24      var
24        nexttermval : strtyp;
24
24      procedure term(var termval : strtyp);
24
24        var
24          nextfacval : strtyp;
24
24        procedure factor(var facval : strtyp);
24          var
24            i : integer;
24
24        begin
24          if nextchar in ['(','-']
25            then
25              begin
```

zu Abb. 1.3.1-11

```
26              case nextchar of
27                '(': begin
28                       readchar;
29                       expr(facval);
30                       if nextchar = ')' then readchar
32                         else writeln('error')
33                     end;
33                '-': begin
34                       readchar;
35                       factor(facval);
36                       gennew(value);
37                       nichtaus(facval,value);
38                       facval:=value
39                     end
39              end
39            end
39          else
39            if nextchar in ['a'..'z','A'..'Z'] then
40            begin
41              for i:=0 to maxstr do facval[i] := ' ';
43              facval[0] := nextchar;
44              readchar
45            end
45        end; { factor }
45
45      begin {term}
45        factor(termval);
46        while nextchar ='*' do begin
48          readchar;
49          factor(nextfacval);
50          gennew(value);
51          undaus(termval,nextfacval,value);
52          termval := value
53          end
53      end; {ende term}
53
53    begin {expr}
53      term(exprval);
54      while nextchar ='+' do begin
56         readchar;
57         term(nexttermval);
58         gennew(value);
59         oderaus(exprval,nexttermval,value);
60         exprval := value
```

zu Abb. 1.3.1-11

```
61          end
61     end; (ende expr)
61
61
61   begin (main)
61
61   writeln(' Konstrukta  Hardwaregenerator Sprache ');
62   writeln(' Version 1.0 810904   R-D. Klein        ');
63   writeln;
64   initnew(value);
65   readchar;
66   expr(wert);
67   writeln(' an ',wert,' liegt ausgang ');
68   end.
```

```
 Konstrukta  Hardwaregenerator Sprache
 Version 1.0 810904   R-D. Klein

e+a*(d+z).
                       ---
d                   ---|>|
                       |=|---&1
z                   ---|1|
                       ---
                       ---
a                   ---| |
                       |&|---&2
&1                  ---| |
                       ---
                       ---
e                   ---|>|
                       |=|---&3
&2                  ---|1|
                       ---
 an &3             liegt ausgang
```

zu Abb. 1.3.1-11

```
 Konstrukta  Hardwaregenerator Sprache
 Version 1.0 810904   R-D. Klein

a*-(h+-k).
                       ---
k                   ---| |O---&1
                       ---
```

```
                 ---
h            ---|>|
                |=|---&2
&1           ---|1|
                 ---

                 ---
&2           ---|  |O---&3
                 ---

                 ---
a            ---|  |
                |&|---&4
&3           ---|  |
                 ---
 an &4            liegt ausgang



 Konstrukta  Hardwaregenerator Sprache
 Version 1.0 810904   R-D. Klein

a+(g*j*r+b+c+j*-(l+-o+i)*m).
                 ---
g            ---|  |
                |&|---&1
j            ---|  |
                 ---
                 ---
&1           ---|  |
                |&|---&2
r            ---|  |
                 ---
                 ---
&2           ---|>|
                |=|---&3
b            ---|1|
                 ---
                 ---
&3           ---|>|
                |=|---&4
c            ---|1|
                 ---
                 ---
o            ---|  |O---&5
                 ---
```

zu Abb. 1.3.1-11

```
                   ---
l             ---|>|
                 |=|---&6
&5            ---|1|
                   ---

                   ---
&6            ---|>|
                 |=|---&7
i             ---|1|
                   ---

                   ---
&7            ---| |0---&8
                   ---

                   ---
j             ---| |
                 |&|---&9
&8            ---| |
                   ---

                   ---
&9            ---| |
                 |&|---&:
m             ---| |
                   ---

                   ---
&4            ---|>|
                 |=|---&;
&:            ---|1|
                   ---

                   ---
a             ---|>|
                 |=|---&<
&;            ---|1|
                   ---
  an &<          liegt ausgang
```

zu Abb. 1.3.1-11

1.3.2 FUNCTION-Deklaration

Funktionen werden an der gleichen Stelle definiert, wie Prozeduren. Wir hatten bisher auch schon von Funktionen Gebrauch gemacht, nämlich von eingebauten Standardfunktionen, wie TRUNC(X) usw. Eine Funktion unterscheidet sich formal von einer Prozedur nur dadurch, daß sie auch einen Wert besitzen kann. Bei Prozeduren ließ sich aber ein Wert an das aufrufende Programm geben, dadurch, daß ein Parameter mit VAR deklariert wurde. Bei Funktionen ist dies eleganter:

entweder: FUNCTION
name : typ ; block

oder: FUNCTION name
parameterliste : typ; block

```
PROGRAM Funktionen(INPUT,OUTPUT);

   VAR
      x1,x2,y1,y2 : REAL;

   FUNCTION distance(x1,y1,x2,y2 : REAL) : REAL;

      BEGIN
         distance := SQRT((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))
      END;

BEGIN (* Hauptprogramm *)
   WRITELN;
   REPEAT
      WRITE(' x1,y1 x2,y2 eingeben : ');
      READ(x1,y1,x2,y2);
      WRITELN;
      WRITELN(' Distanz der beiden Punkte: ',distance(x1,y1,x2,y2));
   UNTIL (x1=x2) AND (y1=y2)
END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
 x1,y1 x2,y2 eingeben : 1,2,3,4

 Distanz der beiden Punkte:  2.82843e0
 x1,y1 x2,y2 eingeben : 1,1,2,2

 Distanz der beiden Punkte:  1.41421e0
 x1,y1 x2,y2 eingeben : 1,1,1,1

 Distanz der beiden Punkte:  0.00000e0
```

Abb. 1.3.2-1
Beispiel Funktionen

Dabei gibt „typ" den Typ des Wertes der Funktion an.
Beispiel:

```
FUNCTION quad(X:REAL):REAL;
   BEGIN
      quad := X·X·X·X
   END;
```

Die Funktion „quad" bildet die vierte Potenz des Wertes des Parameters. Ein Aufruf kann dann wie folgt aussehen:

```
a := quad(b);
```

Der Variablen a wird der Wert von b · b · b · b zugewiesen.

Abb. 1.3.2-1 zeigt ein ausgeführtes Beispiel. Es ist die Aufgabe gestellt, den Abstand zweier Punkte einer Ebene zu berechnen. Dazu wird die Funktion „distance" definiert. Im Hauptprogramm werden die Koordinaten zweier Punkte eingelesen, und die Funktion „distance" zur Berechnung aufgerufen.

Ein weiteres Beispiel zeigt *Abb. 1.3.2-2*. Eine Funktion soll ausgeplottet werden, indem zur Darstellung ein „·"-Zeichen verwendet wird. Es wird die Funktion „poly" definiert, die die Berechnung der gewünschten Funktion beinhaltet. Mit einer Prozedur „plot" wird die Ausgabe der Funktion durchgeführt. Durch die modulare Struktur ist es leicht möglich, beispielsweise die Prozedur „plot" durch eine andere Definition zu ersetzen, um auf einen echten Plotter auszugeben. Auch läßt sich die ge-

```
PROGRAM funktion(INPUT,OUTPUT);

   VAR
      a : REAL;
      i,t1 : INTEGER;

   FUNCTION poly(x:REAL) : REAL;

      BEGIN
         poly := x*x*x - x
      END;

   PROCEDURE plot(x:INTEGER);
      VAR
         i : INTEGER;

      BEGIN
         FOR i:=0 TO x DO
            WRITE(' ');
         WRITELN('*')
      END;

BEGIN
   WRITELN;
   WRITELN(' Funktionsplot ');
   a:=-2;
   REPEAT
      t1 := TRUNC(poly(a)*5+40);
      plot(t1);
      a := a + 0.3
   UNTIL a>=2
END.
```

Abb. 1.3.2-2
Plotten von Funktionen
mit Funktionen

```
    Sorcim PASCAL/M ver 03.05/03.05
     Funktionsplot

*
             *
                       *
                            *
                               *
                               *
                              *
                             *
                            *
                            *
                              *
                                  *
                                          *
                                                      *
```

```
PROGRAM rekursion(INPUT,OUTPUT);

   VAR
      i : INTEGER;

   FUNCTION fac(n : INTEGER) : INTEGER;

      BEGIN
         IF n = 0
            THEN fac := 1
            ELSE fac := fac(n-1)*n
      END;

BEGIN
   WRITELN;
   FOR i:=0 to 7 DO
      WRITELN('FAC von ',i,' ist ',fac(i))
END.

Sorcim PASCAL/M ver 03.05/03.05
FAC von 0 ist 1
FAC von 1 ist 1
FAC von 2 ist 2
FAC von 3 ist 6
FAC von 4 ist 24
FAC von 5 ist 120
FAC von 6 ist 720
FAC von 7 ist 5040
```

Abb. 1.3.2-3 Fakultät Rekursiv

wünschte Funktion leicht durch eine andere ersetzen, ohne im Hauptprogramm viel ändern zu müssen. Wie bei Prozeduren, können auch bei Funktionen mehrere Parameter verwendet oder LOKALE Variablen definiert werden. Rekursion, also der Aufruf einer Funktion durch sich selbst, ist ebenfalls möglich und hierzu gibt es ein klassisches Beispiel, wie *Abb. 1.3.2-3* zeigt. Es gibt eine Funktion, Fakultät genannt, die schon rekursiv definiert ist. FAK(0) ergibt per Definition den Wert 1. Dann gilt ferner FAK(i). Also zu deutsch, die Fakultät eines Wertes ist der Wert multipliziert mit dem Wert der Fakultät des Wertes, vermindert um eins. Beispiel: Zu berechnen ist FAK(3). Also gilt FAK(3) ist FAK(3 – 1) · 3 also FAK(2) · 3, ferner FAK (2 – 1) · 1 · 2 · 3, also FAK(0) · 1 · 2 · 3. Für FAK(0) gilt aber das Abbruchkriterium der Definition, FAK(0) ist 1, also FAK(3) ist 1 · 1 · 2 · 3, also 6. Im Programm ist diese Definitionsweise genau übernommen. Dadurch, das PASCAL die Parameter mit lokalen Variablen verwaltet (wenn kein VAR gegeben ist), funktioniert das Verfahren auch.

1.4 PASCAL und variable Typen

1.4.1 Skalare Typen

Wir kommem nun zu einem ganz neuen Aspekt von PASCAL. Bisher ließen sich die Befehle auch beispielsweise in BASIC noch gerade so lösen. Doch nun zu den besonderen Fähigkeiten von PASCAL, die allerdings auch noch ein paar andere höhere Sprachen haben.

Wir hatten bisher nur die Standard-Typen INTEGER, REAL, CHAR und BOOLEAN gebraucht. Es ist nun aber auch möglich, eigene Typen zu definieren. Dazu gibt es eine Typen-Definition, die sich wie folgt in das Definitionsschema einreiht:

CONST Definitionen
TYPE Definitionen
VAR Definitionen
... wie bisher ...

```
PROGRAM vartyp(INPUT,OUTPUT);
   TYPE
      tag = (montag,dienstag,mittwoch,donnerstag,
             freitag,samstag,sonntag);

   VAR
      tagvar : tag;
      ferientag,arbeitstag : tag;

   BEGIN
      WRITELN;
      ferientag := sonntag;
      FOR tagvar := montag TO sonntag DO
         BEGIN
            WRITE(' TAG ',ORD(tagvar)+1);
            IF tagvar = ferientag
               THEN
                  WRITE(' frei ')
               ELSE
                  WRITE(' nicht frei ');
                  WRITELN
         END
   END.

Sorcim PASCAL/M ver 03.05/03.05
 TAG 1 nicht frei
 TAG 2 nicht frei
 TAG 3 nicht frei
 TAG 4 nicht frei
 TAG 5 nicht frei
 TAG 6 nicht frei
 TAG 7 frei
```

Abb. 1.4.1-1 Variant-Typen

Der einfachste Typ, der definiert werden kann, ist ein scalarer Typ. So läßt sich z. B. einfach angeben:

TYPE
    modell = (rot, gruen, blau);

Es ist damit ein neuer Typ „modell" definiert. Nun kann bei der Deklaration von Variablen auch dieser neue Typ verwendet werden.

VAR
    auto : modell;

Die Variable „auto" ist vom Typ modell, sagt man. sie kann drei Werte annehmen: rot, gruen, blau.

Damit ist z. B. folgender Programmabschnitt möglich

    auto : = rot;
    IF auto = gruen THEN ...

Intern werden der Variablen „auto" INTEGER-Werte zugewiesen. Die angegebenen Werte „rot, gruen, blau" besitzen damit eine Ordnungszahl. Sie wird von links nach rechts, beginnend mit 0 vergeben. Der Wert „rot" besitzt also die Ordnungszahl 0, „gruen" den Wert 1 und „blau" den Wert 2. Die Ordnungszahlen können über die Funktion ORD(×) abgerufen werden, die wir schon früher für CHAR-Größen verwendet hatten. Im Standard-PASCAL ist es nicht möglich, einen solchen Typ direkt auszugeben oder einzulesen, also z. B. „WRITE(auto); ergibt nicht etwa die Ausgabe „rot", sondern führt allenfalls zu einer Fehlermeldung. Mit „WRITE(ORD(auto));" geht es, nur als Ergebnis wird eine Zahl

```
PROGRAM vartyp(INPUT,OUTPUT);

   TYPE
      buchstabe = (alpha,beta,gamma);

   VAR
      bu1 : buchstabe;

BEGIN { Beispiel fuer Operationen }
   WRITELN;
   bu1 := beta;
   WRITELN('ORD',ORD(bu1));
   bu1 := SUCC(bu1);
   WRITELN('SUCC',ORD(bu1));
   bu1 := PRED(PRED(bu1));
   WRITELN('PRED PRED',ORD(bu1));
   IF alpha < beta THEN WRITELN('alpha < beta');
   IF bu1 > alpha THEN WRITELN(' bu1 > alpha ')
END.

Sorcim PASCAL/M ver 03.05/03.05
ORD1
SUCC2
PRED PRED0
alpha < beta
```

Abb. 1.4.1-2
Beispiele für Operationen mit Variant-Typen

```
PROGRAM vartyp(INPUT,OUTPUT);

   TYPE
      zustand = (low,high,tristate);

   VAR
      eingl,enable,ausgang : zustand;
      chl : CHAR;

PROCEDURE buffer(el,enable:zustand; var a:zustand);

BEGIN
   IF el = tristate THEN el := high; { bei TTL }
   IF enable = tristate THEN enable := high;
   IF enable = low
      THEN
         a := el { durchschalten }
      ELSE
         a := tristate
END;

BEGIN { Hauptprogramm }
   WRITELN;
   REPEAT
      WRITE('Eingabe eingl (0 1 T)');
      READ(chl);
      WRITELN;
      CASE chl OF
         '0' : eingl := low;
         '1' : eingl := high;
         'T' : eingl := tristate
      END;
      WRITE('Eingabe enable (0 1 T)');
      READ(chl);
      WRITELN;
      CASE chl OF
         '0' : enable := low;
         '1' : enable := high;
         'T' : enable := tristate
      END;
      buffer(eingl,enable,ausgang);
      CASE ausgang OF
         low : WRITELN(' low ');
         high: WRITELN(' high ');
         tristate: WRITELN(' tristate ')
      END
   UNTIL 1=0
END.
```

Abb. 1.4.1-3
Logiksimulator für
TRI-State-Gatter

```
Sorcim PASCAL/M ver 03.05/03.05
Eingabe eing1 (0 1 T)0
Eingabe enable (0 1 T)1
 tristate
Eingabe eing1 (0 1 T)1
Eingabe enable (0 1 T)0
 high
Eingabe eing1 (0 1 T)0
Eingabe enable (0 1 T)0
 low
Eingabe eing1 (0 1 T)1
Eingabe enable (0 1 T)1
 tristate
Eingabe eing1 (0 1 T)T
Eingabe enable (0 1 T)0
 high
Eingabe eing1 (0 1 T)T
Eingabe enable (0 1 T)T
 tristate
Eingabe eing1 (0 1 T)T
Eingabe enable (0 1 T)1
 tristate
Eingabe eing1 (0 1 T)0
Eingabe enable (0 1 T)T
 tristate
```

zu Abb. 1.4.1-3

ausgegeben. Abhilfe läßt sich hier nur mit einer CASE-Anweisung schaffen. (Es gibt ein Mikroprozessor-PASCAL bei dem es dennoch geht: PASCAL/Z (siehe folgende Kapitel)).

Abb. 1.4.1-1 zeigt ein praktisch ausgeführtes Programm. es wird der Typ „tag" definiert, der die Werte „montag .. sonntag" annehmen kann. „tagvar", „ferientag" und „arbeitstag" sind als Variable vom Typ „tag" definiert. Das Programm zeigt nun, wie elegant mit diesen Variablen umgegangen werden kann.

Neben ORD(X) gibt es aber auch noch andere Funktionen, die verwendet werden können. SUCC(X) gibt den Wert des nächsten Elements und PRED(X) ergibt den Vorgänger, bezogen auf die Reihenfolge der Elemente, wie sie in der TYPE-Definition angegeben wurde. *Abb. 1.4.1-2* zeigt ein Beispiel dafür.

Eine Lösung für ein völlig anderes Problem zeigt *Abb. 1.4.1-3*. Ein TRI-State Buffer soll simuliert werden. Es sind drei Logizustände erlaubt. LOW, HIGH und TRISTATE. Dazu wird der Typ „zustand" definiert. Mit der PROCEDURE „buffer" wird das logische Verhalten des Buffers simuliert. Dabei wird berücksichtigt, daß bei Standard-TTL Logik ein offener Eingang wie mit HIGH beschaltet aussieht. Im Hauptprogramm ist es nun möglich, den Eingang und den Freigabeeingang mit einem der drei Werte zu beschalten und der Wert des Ausgangs wird angegeben.

1.4.2 Bereiche

Durch zwei Konstanten kann ein Bereich definiert werden. Zum Beispiel:

```
TYPE
    index = 1 .. 100;
```

Der Typ Index wird mit einem Bereich von 1 bis 100 definiert. Einer Variablen, die vom Typ „index" ist, kann nur ein Wert zwischen 1 und 100 zugewiesen werden. Liegt der Wert außerhalb des Bereichs, so wird eine Fehlermeldung ausgegeben. Das es sich um einen Bereich handelt, wird durch die zwei Punkte angegeben. *Abb. 1.4.2-1* zeigt ein Programmbeispiel. Der Typ „buchstabe" ist mit einem Bereich von 'A' bis 'Z' definiert, und „ziffer" mit einem Bereich von 0 bis 9. Bei dem Hauptprogramm wird nun abwechselnd ein Buchstabe und eine Ziffer als Eingabe verlangt. Bei einer Fehleingabe gibt es eine Compilerabhängige Fehlermeldung. Normalerweise sollte eine Fehleingabe per Programm

```
PROGRAM subrange(INPUT,OUTPUT);

   TYPE
      buchstabe = 'A' .. 'Z';
      ziffer = '0' .. '9';
      index = 1..100;

   VAR
      i : index;
     bu : buchstabe;
     zi : ziffer;

BEGIN
   WRITELN;
   FOR i:=1 TO 10 DO
   BEGIN
      WRITE(' Buchstabe eingeben ');
      READ(bu);
      WRITELN;
      WRITE(' Ziffer eingeben ');
      READ(zi);
      WRITELN;
      WRITELN('->',bu,zi,'<-',i)
   END
END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
 Buchstabe eingeben B
 Ziffer eingeben 5
->B5<-1
 Buchstabe eingeben J
 Ziffer eingeben 3
->J3<-2
 Buchstabe eingeben K
 Ziffer eingeben 2
->K2<-3
 Buchstabe eingeben M
 Ziffer eingeben 1
->M1<-4
 Buchstabe eingeben A
 Ziffer eingeben 9
->A9<-5
```

Abb. 1.4.2-1 Unterbereiche

```
  Buchstabe eingeben O
  Ziffer eingeben 8
->O8<-6
  Buchstabe eingeben U
  Ziffer eingeben 2
->U2<-7
  Buchstabe eingeben N
  Ziffer eingeben 3
->N3<-8
  Buchstabe eingeben J
  Ziffer eingeben 2
->J2<-9
  Buchstabe eingeben K
  Ziffer eingeben 9
->K9<-10
```

zu Abb. 1.4.2-1

abgeprüft und die automatische Fehlererkennung nur bei der Erkennung von Programmierfehlern verwendet werden.

1.4.3 Mengen

Eine Menge ist eine Ansammlung von Objekten des gleichen Typs. In PASCAL lassen sich nun auch Typen definieren, die eine Menge darstellen können. Beispiel:

```
  TYPE
    bauteile  = (pc,ram,rom,cpu,io,
                 eprom,alu,register,
                 buffer);
    computer = SET OF bauteile;
  VAR
    z80 : computer;
```

Nun, die erste Zeile in der Typ Definition ist ja noch klar. „bauteile“ ist ein Typ, der die Werte „pc.. buffer“ beinhaltet. Der Typ „computer“ definiert nun einen Mengentyp. „z80“ ist als Variable des Typs „computer“ definiert; ihr können also Mengen zugewiesen werden. Beispiel:

```
  z80 := [alu,register,pc];
```

„z80“ ist nun eine Menge von Bauteilen, die „alu“, „register“ und „pc“ beinhaltet.

Mit Mengen kann in PASCAL genauso gerechnet werden, wie es von der Mengenlehre her bekannt ist.

Eine Menge wird durch „[   ]“ angegeben. Es ist möglich, die Vereinigungsmenge, UNION genannt zu bilden. Dies geschieht mit dem „+“ Zeichen:
z80 := [register, alu] + [buffer];
Ebenfalls läßt sich der Durchschnitt bilden. Dazu wird das „·“-Zeichen verwendet:

```
  z80 =: [cpu,register,alu,buffer] ·
         [register,alu,buffer,rom];
```

Mit dem Zeichen „–“ kann die Differenz zweier Mengen gebildet werden, menge2 – menge1 ergibt eine Menge, die alle Werte der menge2 enthält, die in menge2 enthalten waren und nicht in menge1.

Dann gibt es noch Operatoren, die Vergleiche ermöglichen:

menge1 = menge2	Gleichheit zweier Mengen
menge1 <> menge2	Ungleich
menge1 <=menge2	menge1 ist in menge2 enthalten
menge1 >= menge2	menge1 enthält menge2
element IN menge	Das Element „element“ ist in „menge“ enthalten.
[ ]	Leere Menge

```
PROGRAM sets(INPUT,OUTPUT);

   TYPE
      fach = (deutsch,mathe,englisch);
      lehrer = SET OF fach;

   VAR
      anton,mueller,schulze : lehrer;
      klasse : lehrer;
      ch : CHAR;

BEGIN
   WRITELN;
   anton := [mathe,englisch];
   mueller := [deutsch,englisch];
   schulze := [deutsch .. englisch];
   klasse := [];
   REPEAT
      WRITE('Eingeben name (anton,mueller,schulze)');
      READLN(ch);
      CASE ch OF
         'a': klasse := klasse + anton;
         'm': klasse := klasse + mueller;
         's': klasse := klasse + schulze
      END
   UNTIL klasse = [mathe,englisch,deutsch];
   WRITELN(' Klasse bekommt alle Faecher ')
END.
```

Abb. 1.4.3.1
Die Schulklasse
mit Mengen-Befehlen

```
Sorcim PASCAL/M ver 03.05/03.05
Eingeben name (anton,mueller,schulze)schulze
 Klasse bekommt alle Faecher
prun set1
Sorcim PASCAL/M ver 03.05/03.05
Eingeben name (anton,mueller,schulze)mueller
Eingeben name (anton,mueller,schulze)anton
 Klasse bekommt alle Faecher
prun set1
Sorcim PASCAL/M ver 03.05/03.05
Eingeben name (anton,mueller,schulze)mueller
Eingeben name (anton,mueller,schulze)schulze
 Klasse bekommt alle Faecher
```

```
PROGRAM sets(INPUT,OUTPUT);

   TYPE
      element = 'a' .. 'z';
      menge = SET OF element;

   VAR
      menge1,menge2,menge3,grundmenge : menge;

PROCEDURE getmenge(VAR meng : menge);
   VAR
      ch : CHAR;
   BEGIN
      WRITE('Menge a..z .=stop:');
      meng := [];
      REPEAT
         READ(ch);
         IF ch IN grundmenge THEN meng := meng + [ch]
      UNTIL ch = '.';
      WRITELN;
   END;


PROCEDURE mengeaus(mengenvar : menge);
   VAR
      count : element;
   BEGIN
      FOR count := 'a' TO 'z' DO
         IF count IN mengenvar
            THEN WRITE(count)
            ELSE WRITE(' ');
      WRITELN
   END;

BEGIN
   WRITELN;
   grundmenge := ['a' .. 'z'];
   REPEAT
      getmenge(menge1);
      getmenge(menge2);
      WRITELN('menge . --------------------------');
      WRITE('menge1   ');
      mengeaus(menge1);
      WRITE('menge2   ');
```

Abb. 1.4.3-2 Programm für Mengenlehre mit Mengen-Befehlen

```
mengeaus(menge2);
WRITELN('          ---------------------------');
menge3 := menge1 + menge2;
WRITE(' +       ');
mengeaus(menge3);
menge3 := menge1 * menge2;
WRITE(' *       ');
mengeaus(menge3);
menge3 := menge1 - menge2;
WRITE(' -       ');
mengeaus(menge3);
menge3 := grundmenge-menge1;
WRITE('G-menge1');
mengeaus(menge3);
WRITELN('          ---------------------------');
IF menge1 >= menge2
    THEN WRITELN('menge1 einhaelt menge2');
IF menge1 <= menge2
    THEN WRITELN('menge1 ist in menge2 enthalten');
IF menge1 = menge2
    THEN WRITELN('menge1 ist gleich menge2');
IF menge1 <> menge2
    THEN WRITELN('menge1 ist ungleich menge2');
IF menge1 = []
    THEN WRITELN('menge1 ist leer');
IF menge2 = []
    THEN WRITELN('menge2 ist leer');
   UNTIL (menge1=[]) AND (menge2=[])
END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
Menge a..z .=stop:a.
Menge a..z .=stop:abcd.
menge . ---------------------------
menge1  a
menge2  abcd
        ---------------------------
 +      abcd
 *      a
 -
G-menge1 bcdefghijklmnopqrstuvwxyz
        ---------------------------
```

zu Abb. 1.4.3-2

```
menge1 ist in menge2 enthalten
menge1 ist ungleich menge2
Menge a..z .=stop:abcd.
Menge a..z .=stop:b.
menge . -------------------------
menge1  abcd
menge2   b
        -------------------------
 +      abcd
 *       b
 -      a cd
G-menge1    efghijklmnopqrstuvwxyz
        -------------------------
menge1 einhaelt menge2
menge1 ist ungleich menge2
Menge a..z .=stop:meier.
Menge a..z .=stop:mueller.
menge . -------------------------
menge1      e   i   m     r
menge2      e      lm     r  u
        -------------------------
 +          e   i  lm     r  u
 *          e       m     r
 -              i
G-menge1abcd fgh jkl nopq stuvwxyz
        -------------------------
menge1 ist ungleich menge2
Menge a..z .=stop:meier.
Menge a..z .=stop:ei.
menge . -------------------------
menge1      e   i   m     r
menge2      e   i
        -------------------------
 +          e   i   m     r
 *          e   i
 -                  m     r
G-menge1abcd fgh jkl nopq stuvwxyz
        -------------------------
menge1 einhaelt menge2
menge1 ist ungleich menge2
Menge a..z .=stop:the quick brown fox.
Menge a..z .=stop:jumps over the lazy dog.
menge . -------------------------
```

zu Abb. 1.4.3-2

```
menge1   bc ef hi k  no qr tu wx
menge2  a  de gh j lm op rstuv  yz
        --------------------------
 +      abcdefghijklmnopqrstuvwxyz
 *          e  h      o  r tu
 -       bc  f  i k  n  q      wx
G-menge1a  d  g  j lm  p  s  v  yz
        --------------------------
menge1 ist ungleich menge2



Sorcim PASCAL/M ver 03.05/03.05
Menge a..z .=stop:max und fritz jagen.
Menge a..z .=stop:.
menge . --------------------------
menge1  a  defg ij  mn   r tu  x z
menge2
        --------------------------
 +      a  defg ij  mn   r tu  x z
 *
 -      a  defg ij  mn   r tu  x z
G-menge1 bc    h  kl  opq s  vw y
        --------------------------
menge1 einhaelt menge2
menge1 ist ungleich menge2
menge2 ist leer
Menge a..z .=stop:max und fritz jagen kleines vieh.
Menge a..z .=stop:.
menge . --------------------------
menge1  a  defghijklmn   rstuv x z
menge2
        --------------------------
 +      a  defghijklmn   rstuv x z
 *
 -      a  defghijklmn   rstuv x z
G-menge1 bc            opq      w y
        --------------------------
menge1 einhaelt menge2
menge1 ist ungleich menge2
menge2 ist leer
Menge a..z .=stop:max und fritz jagen kleines vieh obwohl.
Menge a..z .=stop:.
menge . --------------------------
```

zu Abb. 1.4.3-2

```
menge1   ab defghijklmno  rstuvwx z
menge2
         --------------------------
  +      ab defghijklmno  rstuvwx z
  *
  -      ab defghijklmno  rstuvwx z
G-menge1   c              pq      y
         --------------------------
menge1 einhaelt menge2
menge1 ist ungleich menge2
menge2 ist leer
Menge a..z .=stop:.
Menge a..z .=stop:.
menge . --------------------------
menge1
menge2
         --------------------------
  +
  *
  -
G-menge1abcdefghijklmnopqrstuvwxyz
         --------------------------
menge1 einhaelt menge2
menge1 ist in menge2 enthalten
menge1 ist gleich menge2
menge1 ist leer
menge2 ist leer
```

zu Abb. 1.4.3-2

Zu IN noch ein paar Worte. Damit ist es möglich zu prüfen, ob ein bestimmtes Element in einer menge enthalten ist. Beispiel:

cpu IN z80 ergibt den Wert FALSE
io IN [ram,rom,io,cpu] ergibt TRUE
alu IN z80 ergibt TRUE

Falls z80 = [alu,buffer,pc] z. B.

Intern werden die Mengenvariablen meist als Bitmap geführt. Das heißt für jedes mögliche element einer Menge ist mindestens ein Bit-Speicherplatz reserviert. Wird das Element einer Menge zugewiesen, so wird das dazugehörige Bit gesetzt.

Abb. 1.4.3-1 zeigt ein Beispiel für das Arbeiten mit den Mengen-Befehlen. Es wird die Aufgabe gestellt, den Unterricht in einer Klasse einzuteilen. Es gibt drei Lehrer: Anton, Müller und Schulze. Dann gibt es noch drei Fächer: Deutsch, Mathe und Englisch, die unterrichtet werden müssen. Jeder Lehrer kann nun bestimmte Fächer unterrichten.

Anton unterrichtet Mathe und Englisch, Müller Deutsch und Englisch und Schulze alle drei Fächer. Das Programm soll nun bei der Eingabe von Lehrern entscheiden, ob eine Klasse mit den angegebenen Fächern auch in allen drei Fächern unterrichtet werden kann

Abb. 1.4.3-2 zeigt noch ein anderes Programm zur Mengenlehre. Als Elemente können Werte von 'a' bis 'z' eingegeben werden. Es werden zwei Mengen eingegeben und dann die verschiedenen Verknüpfungen ausgeführt. Vielen ist vielleicht der Satz „THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG" bekannt. Er hat die besondere eigenschaft alle Buchstaben des Alphabets zu enthalten. Zur Überprüfung ist dieser Satz auch einmal in unser Programm als Menge eingegeben worden. Nun läßt sich mit dem Programm auch ein solcher Satz finden. Es wird ein Satz immer um ein Wort ergänzt, bis die eingegebene Menge mit der Grundmenge übereinstimmt. Als Beispiel wurde hier ein Satz „MAX UND FRITZ JAGEN KLEINES VIEH OBWOHL" gefunden, bei dem nur noch die Buchstaben „C P QY" fehlen.

Vielleicht findet der Leser einen passenden Schluß dieses Satzes mit den fehlenden Buchstaben...

1.5 Strukturierte Typen

1.5.1 Felder

Bisher hatten wir nur Variable mit einer Komponenten definieren können. Nehmen wir einmal an, es ist Aufgabe 20 Werte einzulesen, danach eine Kurve der Werte auszugeben und noch ein paar Berechnungen mit den Werten durchzuführen. Dann wäre es sehr umständlich für jeden einzulesenden Wert, eine andere Variable verwenden zu müssen. Immer wenn mehrere Werte von einem Typ vorhanden sein sollen, ist es angebracht, ein Feld zu verwenden. Über einen Index kann dann eine Komponente des Feldes angesprochen werden. Beispiel:

```
VAR
   werte : ARRAY [1 .. 20] OF REAL;
```

Es wird ein Feld des Namens „werte" definiert, das 20 Komponenten des Typs REAL umfaßt. Der Zugriff auf ein Element erfolgt dann z. B. wie folgt:

```
werte[3]
```

Hier wird das dritte Element angesprochen. Zuweisungen sind dann wie gewohnt ausführbar. *Abb. 1.5.1-1* zeigt ein Beispiel für die Handhabung mit einem Feld. Als Index kann auch eine Variable stehen, die mit dem Typ bei der Bereichsdefinition ([1..20]) übereinstimmen muß. Dabei sind nur skalare Typen, also z. B. nicht etwa REAL, zugelassen. Eine Möglichkeit wäre z. B. auch:

```
TYPE
   preise = ( dm , dollar );
   ware = ARRAY [ preise ] OF REAL;
VAR
cpu : ware;
```

Mit cpu[dm] wird die eine Komponente angegeben, mit cpu[dollar] die andere Komponente der Variablen „cpu". Nun, hier wäre natürlich ein Feld mit zwei Indizes schön, um z. B. ein Warenfeld aufzubauen, das aus zwei Komponenten besteht. Auch dies ist möglich. Dafür gibt es zwei Wege:

```
warengruppe = ARRAY [preise]
              OF ARRAY[1..500]
              OF REAL;
```

oder

```
warengruppe = ARRAY [preise,
              1..500] OF REAL;
```

Eine Variable, die mit „warenfeld : warengruppe" definiert ist, kann über Doppelindex adressiert werden. Also z. B.

```
writeln(warenfeld[dm, 10])
```

gibt die 10te Komponente von „warenfeld" mit der Bedingung „dm" aus. Dort können z. B. die DM Beträge gespeichert sein. Die andere Komponente, die mit „dollar" angesprochen wird, kann z. B.

```
PROGRAM ara(INPUT,OUTPUT);

   VAR
      feld : ARRAY [1..10] OF REAL;
      temp : REAL;
      i : INTEGER;

BEGIN
   WRITELN;
   FOR i := 1 TO 10 DO
      BEGIN
         WRITE('Feld [',i,'] eingeben ');
         READ(feld[i]);
         WRITELN
      END;
   temp := 0;
   FOR i:=1 to 10 DO temp := temp + feld[i];
   WRITELN(' Mittelwert ',temp/10)
END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
Feld [1] eingeben 45.65
Feld [2] eingeben 8.3
Feld [3] eingeben 3
Feld [4] eingeben 90
Feld [5] eingeben 100
Feld [6] eingeben 78
Feld [7] eingeben 65.23
Feld [8] eingeben 34.2
Feld [9] eingeben 23.45
Feld [10] eingeben 55.33
 Mittelwert  5.03160e1
```

Abb. 1.5.1-1 Eindimensionale Felder

die US-Preise beinhalten. *Abb. 1.5.1-2* zeigt ein Beispiel für den Gebrauch des Doppelindex. Ein zweidimensionales Feld wird eingelesen und anschließend wieder ausgegeben.

1.5.2 Gepackte Felder

Wenn wir ein Feld des Typs BOOLEAN oder CHAR definieren, so belegt ein Element des Feldes normalerweise ein Maschinenwort. Es gibt in PASCAL eine Zusatzanweisung, die es erlaubt, derartige Felder weniger Speicherintensiv aufzubauen. Dazu wird das Wort PACKED verwendet. Beispiel:

```
VAR
   bitmap : PACKED ARRAY[1..40]
   OF BOOLEAN;
```

```
PROGRAM ara(INPUT,OUTPUT);
   TYPE
      index = 1..3;

   VAR
      feld : ARRAY [index,index] OF INTEGER;
      i,j : index;

BEGIN
   WRITELN;
   WRITELN('Eingabe des Feldes');
   FOR i:=1 TO 3 DO
      FOR j:=1 TO 3 DO
         BEGIN
            WRITE('feld[',i,',',j,']:');
            READ(feld[i,j])
         END;
   WRITELN;
   FOR i:=1 TO 3 DO
      BEGIN
         FOR j:=1 TO 3 DO
            WRITE(feld[i,j],' ');
         WRITELN
      END
END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
Eingabe des Feldes
feld[1,1]:1
feld[1,2]:2
feld[1,3]:3
feld[2,1]:4
feld[2,2]:5
feld[2,3]:6
feld[3,1]:7
feld[3,2]:8
feld[3,3]:9

1 2 3
4 5 6
7 8 9
```

Abb. 1.5.1-2 Zweidimensionale Felder

Die Variable „bitmap“ ist nun ein solch gepacktes Feld. Der Nachteil von solchen Feldern ist eine durchschnittlich längere Ausführungszeit, da die einzelne Komponente des Feldes erst berechnet werden muß. Der Vorteil ist ein geringerer Speicherbedarf. Wurde das Feld „bitmap“ z. B. nicht gepackt, so belegt es 40 Maschinenworte. Nimmt man eine 16-Bit-Maschine, so wird beim gepackten Feld nur ein Platz von drei Worten in Anspruch genommen. Um zu schnellen Berechnungen ein solches Feld, in ein anderes, nichtgepacktes und umgekehrt kopieren zu können, gibt es im Standard PASCAL die Befehle PACK und UNPACK.

```
PACK(nichtpack,displacement,
  packfeld);
UNPACK(packfeld,nichtpackfeld,
  displacement);
```

Dabei gilt wenn:

```
packfeld : PACKED ARRAY
  [u..v] OF t
```

und

```
nichtpackfeld : ARRAY [m..n] OF t
```

```
PACK(n,d,p) ist
  FOR j:=u TO v DO
    p[j]:=n[j-u+d]
UNPACK(p,n,d,) ist
  FOR j:=u TO v DO
    n[j-u+d]:=p[j]
```

Bei vielen Mikrorechner-Implementationen ist PACK und UNPACK nicht verfügbar.

1.5.3 Zeichenketten

In der WRITE-Anweisung hatten wir schon Zeichenketten (Strings) verwendet. Es gibt aber auch noch andere Möglichkeiten:

```
CONST
  meldung = 'Bitte Dateneingabe->';
```

Es wird die Konstante „meldung“ definiert. Eine Anweisung:

```
WRITE(meldung);
```

gibt z. B. diesen Text aus.

Um Stringvariable zu definieren muß ein Feld verwendet werden. Dies geschieht dann wie folgt:

```
VAR
  string : PACKED ARRAY
  [1..80] OF CHAR;
```

„string“ wird hier als gepacktes Feld definiert, um eine Reihe von Vergleichsoperationen zu ermöglichen, die nur auf gepackte Felder des Typs CHAR anwendbar sind. Der Stringvariablen können nun auch Texte direkt zugewiesen werden, jedoch muß die Länge exakt mit der Länge des Feldes übereinstimmen. Beispiel:

```
VAR
  string1 : PACKED ARRAY
  [1..6] OF CHAR;
```

Dann ist es möglich zu schreiben:

```
string1 := 'Text  '
```

Natürlich kann nach wie vor auch über einen Index auf Komponenten des Feldes zugegriffen werden.

Wie schon gesagt, sind auch Vergleiche möglich. Dabei müssen die zu vergleichenden Felder erstens vom gleichen Typ sein, also gleiche Länge besitzen und außerdem auch gepackt dargestellt werden. *Abb. 1.5.3-1* zeigt ein Programmbeispiel dazu. Das Einlesen der Strings geschieht über ein Unterprogramm mit dem Namen „lesein“, da es nicht möglich ist, READ(string1) zu schreiben.

1.5.4 Datensätze (Records)

Records sind ebenfalls strukturierte Datentypen. Ein Record besteht aus mehreren Komponenten, ähnlich wie bei

```
PROGRAM strara(INPUT,OUTPUT);

   CONST
      meldung = 'Strings ';
   TYPE
      karte = PACKED  ARRAY [1..80] OF CHAR;
      meldvar = PACKED ARRAY [1..6] OF CHAR;
   VAR
      feld1,feld2 : karte;
      strzuw : meldvar;

PROCEDURE lesein(VAR str : karte);
   VAR
      i : INTEGER;
      ch : CHAR;
   BEGIN
      FOR i:=1 TO 80 DO str[i] := ' ';
      i:=1;
      REPEAT
         READ(ch);
         str[i] := ch;
         i := i + 1;
      UNTIL EOLN OR (i=81);
      WRITELN;
   END;


BEGIN
   WRITELN;
   WRITELN(meldung);
   strzuw := 'STRING';
   WRITELN(strzuw);
   WRITE('string1 eingeben ');
   lesein(feld1);
   WRITE('string2 eingeben ');
   lesein(feld2);
   WRITELN;
   WRITELN(feld1);
   WRITELN(feld2);
   IF feld1 < feld2 THEN WRITELN('string1 vor string2');
   IF feld1 > feld2 THEN WRITELN('string2 vor string1');
   IF feld1 = feld2 THEN WRITELN('Gleiche Strings ');
END.
```

Abb. 1.5.3-1 String-Felder

```
Sorcim PASCAL/M ver 03.05/03.05
Strings
STRING
string1 eingeben test name1

string2 eingeben test name2


test name1

test name2

string1 vor string2
```

```
Sorcim PASCAL/M ver 03.05/03.05
Strings
STRING
string1 eingeben gleich

string2 eingeben gleich


gleich

gleich

Gleiche Strings
```

```
PROGRAM rec(INPUT,OUTPUT);

   TYPE
      bauteil =
         RECORD
            name : PACKED ARRAY[1..10] OF CHAR;
            nr : INTEGER;
            stueck : INTEGER;
            preis : REAL
         END;

   VAR
      ic1,ic2 : bauteil;

BEGIN
   WRITELN;
   ic1.name := '74 LS 00   ';
   ic1.nr := 1;
   ic1.stueck := 10;
   ic1.preis := 1.05;
   ic2 := ic1;
   WRITELN(ic2.name);
   WRITELN(ic2.nr);
   WRITELN(ic2.stueck);
   WRITELN(ic2.preis)
END.


Sorcim PASCAL/M ver 03.05/03.05
74 LS 00
1
10
 1.05000e0
```

Abb. 1.5.4-1
Datensätze (Records)

den Feldern. Im Gegensatz zu den Feldern, müssen jedoch die einzelnen Komponenten eines Records nicht vom selben Typ (REAL, INTEGER...) sein. Die Komponenten werden auch nicht über einen numerischen Index angesprochen, sondern durch einen symbolischen Namen. Records werden bei der TYPE-Definition angegeben.

Beispiel:

```
TYPE
   bauteil =
      RECORD
         anzahl : INTEGER;
         preis : REAL
      END;
```

Es wird damit der Typ „bauteil" definiert. Eine Variable, die von diesem Typ

ist, besitzt zwei Komponenten
Beispiel:

```
VAR
    ic : bauteil;
```

Die Variable „ic" besitzt eine Komponente, die mit „anzahl" bezeichnet wurde und eine INTEGER-zahl aufnehmen kann, und aus einer Komponenten „preis", die eine REAL-Zahl beinhaltet. Der Zugriff auf die Komponenten erfolgt durch Angabe des Komponentennamens, der dem Namen der Variablen durch einen Punkt getrennt, nachgestellt wird.

Mit „ic.anzahl" ist die erste Komponente erreichbar, mit „ic.preis" die zweite Komponente der Variablen „ic".

Abb. 1.5.4-1 zeigt ein kurzes Programm, das von der RECORD-Definition Gebrauch macht. Es zeigt, wie Zuweisungen und Ausgaben mit Records möglich sind.

Interessant wird die anwendung von Records bei der zusätzlichen Verwendung von Feldern. Es ist möglich, ein Feld zu vereinbaren, das aus Komponenten besteht, die aus Records bestehen. Ebenfalls können innerhalb von Records wieder Felder oder Records verwendet werden. Damit kann man recht komplexe Strukturen definieren. Wird in einem kurzen Programmstück dieselbe Recordvariable verwendet, so kann mit Hilfe der WITH-Anweisung die Schreibweise vereinfacht werden. Es ist dann möglich, die Komponenten einer Variablen nur durch Angabe der Komponentennamen zu erreichen.
Beispiel:

```
WITH ic DO
    anzahl := 2;
```

ist gleichwertig zu:

```
    ic.anzahl := 2;
```

Dabei kann nach dem Wort DO ein Block folgen. Der Vorteil ist erst bei komplexeren Strukturen sichtbar, doch sollte man sich vor einer zu häufigen Verwendung hüten, da Programme durch Verwendung der WITH-Anweisung auch unübersichtlich werden können.

Abb. 1.5.4-2 zeigt ein Beispiel für eine Lagerverwaltung. Das Lager soll verschiedene Bauteile (hier maximal 10) aufnehmen. Jedes Bauteil hat einen Namen und einen Preis. Ferner ist eine bestimmte Anzahl von dem jeweiligen Bauteil vorhanden.

Nach Start des Programms wird ein Menü ausgegeben. Es können dann verschiedene Befehle gegeben werden. Der Lagerbestand kann mit dem Befehl „l" ausgegeben werden. Dann kann ein Bauteil mit dem Befehl „e" eingegeben werden. Ferner kann die Lagerbestandssumme mit dem Befehl „s" angezeigt werden. Mit „f" wird der Programmlauf beendet.

Um exakte Werte bei der Rechnung mit Preisen zu erhalten, wurde hier nicht die REAL-Darstellung verwendet, sondern ein neuer Typ mit dem Namen „spezreal" definiert. Dieser kann eine Zahl aufnehmen, die in einen ganzen und gebrochenen Teil geteilt ist. Dabei werden diese beiden Teile mit INTEGER-Größen dargestellt. Ein weiterer Aspekt bei Records sind die sogenannten Variantrecords. Wir hatten bisher für jede Komponente genau einen Speicherplatz belegt. Es ist aber auch möglich, verschiedene Typen in einem Gebiet unterzubringen. In Abhängigkeit von einem anzugebenden Begriff wird dann der Inhalt einer Speicherzelle unterschiedlich interpretiert. Zur Definition von Variant-Typen wird die CASE-Anweisung verwendet, die hier jedoch eine andere Bedeutung besitzt, als wir es bisher gewohnt waren.

```
PROGRAM lager(INPUT,OUTPUT);

   TYPE
      spezreal =
         RECORD
            ganz : INTEGER;
            gebr : INTEGER (dm, pf)
         END;

      bauteil =
         RECORD
            name : PACKED ARRAY[1..10] OF CHAR;
            stueck : INTEGER;
            preis : spezreal
         END;
      bestand = ARRAY [1..10] OF bauteil;
   VAR
      teile : bestand;
      ch1 : CHAR;
      t : spezreal;
      i : INTEGER;

PROCEDURE getpreis(VAR t : spezreal);
   VAR
      ch : CHAR;
      i : INTEGER;
   BEGIN { lesen format xx.xx oder xx }
      t.ganz := 0;
      t.gebr := 0;
      REPEAT
         READ(ch);
         IF ch IN ['0' .. '9'] THEN
            t.ganz := t.ganz*10+ORD(ch)-ORD('0')
      UNTIL NOT(ch IN ['0' .. '9']);
      IF ch = '.' THEN
         BEGIN
            READ(ch);
            IF ch IN ['0' .. '9'] THEN
               BEGIN
                  t.gebr := 10*(ORD(ch)-ORD('0'));
                  READ(ch);
                  IF ch IN ['0' .. '9'] THEN
                     t.gebr := t.gebr + ORD(ch)-ORD('0')
               END
```

Abb. 1.5.4-2
Lagerverwaltung
mit Records

```
         END;
      WRITELN
   END;
PROCEDURE auspreis(t : spezreal);
   BEGIN
      WRITE(t.ganz,'.');
      IF t.gebr<=9
         THEN WRITE('0',t.gebr)
         ELSE WRITE(t.gebr)
   END;


PROCEDURE liste(teil : bestand);
   VAR
      i : INTEGER;
   BEGIN
      WRITELN('index name            anz   preis');
      FOR i:=1 TO 10 DO
         WITH teil[i] DO
            BEGIN
               WRITE(i:5,' ',name,' ',stueck:3,'     ');


               auspreis(preis);
               WRITELN
            END;
   END;

PROCEDURE clr(VAR teil : bestand);
   VAR
      i : INTEGER;
   BEGIN
      FOR i:=1 TO 10 DO
         WITH teil[i] DO
            BEGIN
               name := '               ';
               stueck:=0;
               preis.ganz := 0;
               preis.gebr := 0
            END
   END;

PROCEDURE einles(VAR teil : bestand;i : INTEGER);
   VAR
      j : INTEGER;
      ch : CHAR;
```

zu Abb. 1.5.4-2

```
   BEGIN
      WITH teil[i] DO
         BEGIN
            WRITE('name:');
            j := 1;
            REPEAT
               READ(ch);
               name[j] := ch;
               j := j + 1
            UNTIL EOLN OR (j=11);
            WRITE('Stueckzahl:');
            READLN(stueck);
            WRITE('Preis:');
            getpreis(preis)
         END
   END;

PROCEDURE summe(teil : bestand;VAR sum : spezreal);
   VAR
      i,j : INTEGER;
   BEGIN
      sum.ganz :=0;
      sum.gebr :=0;
      FOR i:=1 TO 10 DO
         WITH teil[i] DO
            FOR j:=1 TO stueck DO
               BEGIN
                  sum.gebr := sum.gebr + preis.gebr;
                  IF sum.gebr >=100 THEN
                     BEGIN
                        sum.gebr := sum.gebr -100;
                        sum.ganz := sum.ganz + 1
                     END;
                  sum.ganz := sum.ganz + preis.ganz
               END
   END;

BEGIN
   WRITELN;
   clr(teile);
   WRITELN('Lagerhaltung ');
   REPEAT
      WRITELN('l = listen');
      WRITELN('e = einlesen');
```

zu Abb. 1.5.4-2

```
  WRITELN('s = bestandssumme');
  WRITELN('f = final  ');
  WRITE('------>');
  READ(ch1);
  WRITELN;
  IF ch1 IN ['l','e','s'] THEN
     CASE ch1 OF
        'l': liste(teile);
        's': BEGIN
                summe(teile,t);
                WRITELN('Summe:',t.ganz,'.',t.gebr,' DM'
             END;
         'e': BEGIN
                 WRITE('Index: ');
                 READ(i);
                 einles(teile,i)
              END
     END;
   UNTIL ch1 = 'f'
END.
```

zu Abb. 1.5.4-2

```
Sorcim PASCAL/M ver 03.05/03.05
Lagerhaltung
l = listen
e = einlesen
s = bestandssumme
f = final
------>l
index name          anz   preis
    1                0     0.00
    2                0     0.00
    3                0     0.00
    4                0     0.00
    5                0     0.00
    6                0     0.00
    7                0     0.00
    8                0     0.00
    9                0     0.00
   10                0     0.00
l = listen
e = einlesen
s = bestandssumme
f = final
------>e
Index: 1
name:7400
Stueckzahl:12
Preis:0.45
```

```
l = listen
e = einlesen
s = bestandssumme
f = final
------>e
Index: 2
name:z80
Stueckzahl:2
Preis:29.90
l = listen
e = einlesen
s = bestandssumme
f = final
------>l
index name          anz   preis
    1 7400           12    0.45
    2 z80             2   29.90
    3                 0    0.00
    4                 0    0.00
    5                 0    0.00
    6                 0    0.00
    7                 0    0.00
    8                 0    0.00
    9                 0    0.00
   10                 0    0.00
```

```
l = listen
e = einlesen
s = bestandssumme
f = final
------>s
Summe:65.20 DM
l = listen
e = einlesen
s = bestandssumme
f = final
------>e
Index: 3
name:7403
Stueckzahl:34
Preis:0.67
l = listen
e = einlesen
s = bestandssumme
f = final
------>l
```

```
index name          anz  preis
    1 7400           12   0.45
    2 z80             2  29.90
    3 7403           34   0.67
    4                 0   0.00
    5                 0   0.00
    6                 0   0.00
    7                 0   0.00
    8                 0   0.00
    9                 0   0.00
   10                 0   0.00
l = listen
e = einlesen
s = bestandssumme
f = final
------>s
Summe:87.98 DM
l = listen
e = einlesen
s = bestandssumme
f = final
------>f
```

zu Abb. 1.5.4-2

```
PROGRAM variant(INPUT,OUTPUT);

   TYPE
      multi = (int,re,ch);

      intrech =
         RECORD
            CASE form : multi OF
               int : (intval : INTEGER);
               re : (realval : REAL);
               ch : (charval : CHAR)
         END;
   VAR
      multi1 : intrech;

BEGIN
   WRITELN;
   multi1.form := int;
   multi1.intval := 10;
   WRITELN(multi1.intval);
   multi1.form := ch;
   multi1.charval := '*';
   WRITELN(multi1.charval);
   WRITELN(multi1.intval)
END.
```

```
Sorcim PASCAL/M ver 03.05/03.05
10
*
42
```

Abb. 1.5.4-3 Variant-Records

Beispiel:

```
TYPE
  entscheidung = (erste,zweite);
  doppel =
    RECORD
      CASE welche :
      entscheidung OF
        erste:
          (name1 : INTEGER);
        zweite:
          (name2 : REAL)
    END;
```

Die Komponente „welche“ von „doppel“ gibt an, welches Teilfeld in der CASE-Anweisung gültig ist. Es wird dabei auch durch die Namen „name1“ und „name2“ bestimmt. Variablen, die vom Typ „doppel“ sind, belegen nur den Speicherplatz der Komponente „welche“ und den Speicherplatz einer der beiden Komponenten „name1“ oder „name2“. Belegt „name2“ z. B. mehr Platz als „name1“, so wird der Platz für „name2“ verwendet.

Abb. 1.5.4-3 zeigt ein Beispiel für den Gebrauch dieser Konstruktion.

Eine Variant-Definition kann auch ohne Angabe einer eigenen Komponenten für die Komponentenauswahl angegeben werden. *Abb. 1.5.4-4* zeigt ein Beispiel dazu. Der Zugriff erfolgt hier ausschließlich durch Angabe des Komponentennamens.

```
PROGRAM variant(INPUT,OUTPUT);

   TYPE
      mehrfach = (int,re);
      intre =
         RECORD
            CASE mehrfach OF
               int : (intval : INTEGER);
               re  : (realval : REAL)
         END;

   VAR
      dop : intre;

BEGIN
   dop.intval := 10;
   WRITELN;
   WRITELN(dop.intval);
   dop.realval := 10;
   WRITELN(dop.realval)
END.


Sorcim PASCAL/M ver 03.05/03.05
10
 1.00000e1
```

Abb. 1.5.4-4
Variant-Records weiteres Beispiel

Die Verwendung dieser Variantstrukturen ist außerordentlich vielseitig, sie sollte jedoch nicht für Programmiertricks verwendet werden, wie dies in unserem Beispiel geschieht. Durch Ausgabe der Komponente „multi1.intval" in Abb. 1.5.4-3, wird ein Wert ausgegeben, der zuvor mit einer CHAR-Größe belegt wurde. Dies funktioniert nicht auf allen Rechnern gleich und ist daher nach Möglichkeit zu unterlassen. Sinnvoll ist die Anwendung der Variantstrukturen zum Einsparen von Speicherplatz in Fällen, wo für eine Variable eine Alternative in der internen Struktur besteht. Z. B. eine Variable, die eine grafische Figur darstellen soll. Die Figur soll dabei von unterschiedlichem Typ sein, z. B. einmal ein Rechteck, durch Angabe der beiden Seiten, zum anderen ein Kreis, wobei Mittelpunkt und Radius angegeben wird. Dann ließe sich folgende Struktur definieren:

```
TYPE
  fall = (rechteck, kreis);
  figure =
    RECORD
      CASE welcher : fall OF
        rechteck: (hoehe: REAL;
                   breite: REAL);
        kreis: (mittex: REAL;
                mittey: REAL;
                radius: REAL)
    END;
    vielfigur = ARRAY [1..100] OF
                figure;
```

Mit

```
VAR
  speicher : vielfigur;
```

wird ein Feld definiert, dessen Komponenten von dem Type „figure" sind. Hier kommt der Vorteil zum Ausdruck. Durch die Verwendung der Variant-Struktur ist ein viel klarerer Programmaufbau möglich.

1.6 Dynamische Datenstrukturen

Bisher hatten wir nur statische Strukturen betrachtet. Der Speicherplatz für Variablen war fest reserviert. Auch Feldgrenzen mußten vor der Ausführung feststehen. Bei manchen Aufgaben ist es aber bei der Programmerstellung nicht klar, wieviel Platz später bei der Ausführung reserviert werden muß. Der Platz kann erst bei der Ausführung des Programms festgelegt werden. In PASCAL ist es nicht möglich, Feldgrenzen mit Variablen anzugeben, die z. B. nach dem Einlesen über eine READ-Anweisung belegt werden. Feldgrenzen müssem immer Konstanten sein. Für die dynamischen Strukturen stehen andere Hilfsmittel zur Verfügung.

1.6.1 Zeiger

Zur Realisierung von dynamischen Strukturen werden in PASCAL Zeiger verwendet. Ein Zeiger, oder auch Pointer genannt, ist ein Typ wie INTEGER oder REAL. Die Deklaration eines Pointertyps erfolgt dabei so:

```
TYPE
  zeiger = ↑objekt
```

Der Typ „zeiger" ist dabei ein Pointer auf „objekt". „objekt" muß dann noch definiert werden. Dabei erfolgt hier ausnahmsweise die Definition nach dem Aufruf. Dies ist nötig, um auch rekursive Strukturen zu ermöglichen.

Für „objekt" kann z. B. gelten:

```
objekt = INTEGER;
```

Wird mit

```
VAR
  zeige : zeiger;
```

die Variable „zeige" definiert, so kann sie auf einen INTEGER Platz zeigen, da „objekt" vom Typ INTEGER ist.

Soll auf ein „objekt" zugegriffen werden, so wird dies durch Anfügen des Zeichens ↑ angegeben.

```
PROGRAM pointer(INPUT,OUTPUT);

    TYPE
        zeiger = ↑objekt;
        objekt = PACKED ARRAY[1..6] OF CHAR;

    VAR
        zei1,zei2 : zeiger;

BEGIN
    WRITELN;
    NEW(zei1);
    zei1↑ := 'NAME1 ';
    NEW(zei2);
    zei2↑ := 'NAME2 ';
    WRITELN(zei1↑);
    WRITELN(zei2↑);
    zei1↑ := zei2↑;
    WRITELN(zei1↑);
    WRITELN(zei2↑);
    zei1↑ := 'TEST12';
    zei2 := zei1;
    WRITELN(zei2↑);
    zei1↑ := 'ZUW1  ';
    WRITELN(zei2↑)
END.

Sorcim PASCAL/M ver 03.05/03.05
NAME1
NAME2
NAME2
NAME2
TEST12
ZUW1
```

Abb. 1.6.1-1 Zeiger

Mit

zeige↑

kann auf den Inhalt von „objekt" zugegriffen werden.

Es müssen nun noch derartige Objekte kreiert werden, so daß es dann möglich ist, damit zu arbeiten. Dazu gibt es die Prozedur NEW(p). Sie erzeugt Platz für ein Objekt von dem Typ, der für „p" zugelassen ist. Die Variable „p" muß dabei ein Pointer sein. Der Pointer zeigt dann sofort auf das erzeugte Objekt. Dies geschieht zur Laufzeit des Programms, aus der sich dynamische Strukturen ergeben. *Abb. 1.6.1-1* zeigt ein kurzes Beispiel für den Gebrauch von Pointern. Vor der Ausführung des ersten Befehls gibt es zwei Pointer, wie sie in *Abb.*

1.6.1-2 dargestellt sind. Dann werden mit NEW(zei1) und NEW(zei2) zwei neue Objekte kreiert. Ihnen werden dabei auch Werte zugewiesen. Die Situation zeigt *Abb. 1.6.1-3.* Mit der Zuweisung „zei1↑:=zei2↑“ wird an das erste Objekt der Wert „NAME2“ zugewiesen. *Abb. 1.6.1-4* zeigt die Objekte. Mit der Anweisung „zei2:=zei1“ wird das Verhältnis in *Abb. 1.6.1-5* erreicht. Auf das zweite Objekt gibt es keinen Verweis mehr, es ist also nicht mehr erreichbar. Im Standard-PASCAL gibt es eine Möglichkeit, derartige belegte Speicherplätze wieder zurückzugewinnen. Dies geschieht mit dem Befehl DISPOSE(pointer). Dazu muß der Befehl aber vor der Zuweisung an „zei2“ ausgeführt werden. Also z. B.:

```
DISPOSE(zei2);
zei2:=zei1;
```

DISPOSE ist aber bei den wenigsten Mikroprozessor-PASCAL-Systemen realisiert. Dort gibt es andere Befehle, die ähnlich wirken (siehe Kapitel 2 ff).

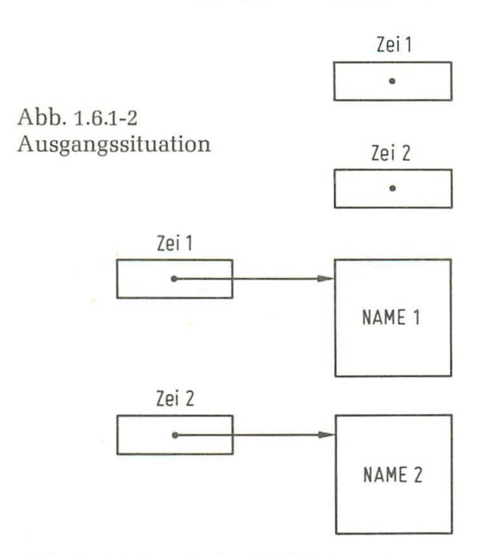


Abb. 1.6.1-2 Ausgangssituation

Abb. 1.6.1-3 Nach den NEW-Anweisungen

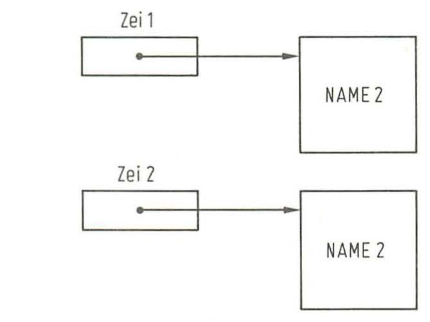


Abb. 1.6.1-4 Zuweisung nach Inhalt

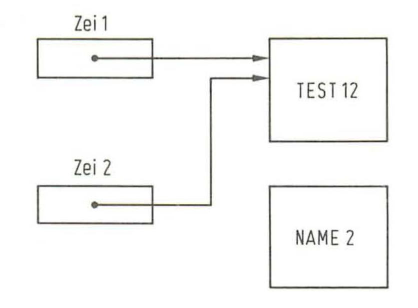


Abb. 1.6.1-5 Zuweisung von Zeigern

1.6.2 Listenstrukturen

Die Art, wie im vorherigen Abschnitt Pointer gebraucht wurden, ist noch nicht ganz sinnvoll. Denn für jedes Objekt haben wir einen eigenen Pointer verbraucht. Sinnvoll wird die Verwendung von Pointern zum Beispiel beim Aufbau einer Listenstruktur. Dabei enthält ein Objekt mehrere Komponenten, mit einer Komponenten die wieder ein Pointer-Typ ist. Diese Komponente kann dann auf ein weiteres Objekt zeigen, dessen Pointer wieder auf ein weiteres Objekt zeigt und so fort. Es wird nun nur noch ein Pointer benötigt, der auf den Start dieser Liste zeigt. Alle Elemente der Liste sind von da aus erreichbar. *Abb. 1.6.2-1* zeigt ein Beispiel für den Aufbau einer Listenstruktur. Das Objekt besteht aus zwei Elementen. Einmal aus einem String, der aus sechs Zeichen besteht, zum anderen aus einem Pointer,

```
PROGRAM link(INPUT,OUTPUT);

   TYPE
      zeiger = ↑objekt;
      objekt =
         RECORD
            nachfolger : zeiger;
            daten : PACKED ARRAY[1..6] OF CHAR;
         END;

   VAR
      zeil,basis : zeiger;

BEGIN
   WRITELN;
   NEW(zeil);
   zeil↑.daten := 'LISTE1';
   basis := NIL;
   zeil↑.nachfolger := basis;
   basis := zeil;
   NEW(zeil); (naechster Eintrag)
   zeil↑.daten := 'LISTE2';
   zeil↑.nachfolger := basis;
   basis := zeil;
   NEW(zeil);
   zeil↑.daten := 'LISTE3';
   zeil↑.nachfolger := basis;
   basis := zeil;
   REPEAT
      WRITELN('Listen Ausgabe ->',zeil↑.daten,'<-');
      zeil := zeil↑.nachfolger
   UNTIL zeil = NIL
END.

Sorcim PASCAL/M ver 03.05/03.05
Listen Ausgabe ->LISTE3<-
Listen Ausgabe ->LISTE2<-
Listen Ausgabe ->LISTE1<-
```

Abb. 1.6.2.1
Listenstruktur

der mit „nachfolger“ bezeichnet ist. Die Variablen „zei1“ und „basis“ sind vom Typ Pointer. Mit NEW(zei1) wird ein Objekt erzeugt. Dann wird die Daten-Komponente mit einem Text vorbelegt. Der Zeiger „basis“ wird mit dem Wert NIL belegt, das heißt, er zeigt auf „nichts“. NIL ist nicht mit 0 zu verwechseln, es

Abb. 1.6.2-3 Trickprogramm

```
PROGRAM pointer(INPUT,OUTPUT);

   TYPE
      zeiger = ↑objekt;
      objekt = INTEGER;
   VAR
      zeil : zeiger;

PROCEDURE druckepointer(pointer : zeiger);
   TYPE
      doppel = (pointtyp,integtyp);
      misch =
         RECORD
            CASE doppel OF
               pointtyp : (pointerval : zeiger);
               integtyp : (integval : INTEGER)
         END;
   VAR
      gemischt : misch;
   BEGIN
      gemischt.pointerval := pointer;
      WRITE('->',gemischt.integval,'<- ')
   END;


BEGIN
   WRITELN;
   NEW(zeil);
   zeil↑ := 10;
   druckepointer(zeil);
   WRITELN(zeil↑);
   NEW(zeil);
   zeil↑ := 20;
   druckepointer(zeil);
   WRITELN(zeil↑)
END.
```

basis Zei 1

LISTE 1 LISTE 2 LISTE 3

Abb. 1.6.2-2 Listenstruktur

```
Sorcim PASCAL/M ver 03.05/03.05
->11696<- 10
->11698<- 20
```

entspricht eher einer leeren Menge. Der Nachfolger des ersten Wertes, der vom Typ Pointer ist, „basis" wird mit dem Wert „zei1" besetzt, so daß nun auf das Objekt, das zuvor kreiert wurde, gezeigt wird. Nun kann mit NEW(zei1) ein neues Objekt kreiert werden. Der Nachfolger dieses Objekts erhält den Wert von „basis" und zeigt damit auf das zuvor erzeugte Objekt. Nach Anlegen einer solchen Liste zeigt „basis" immer auf das aktuelle Element. „zei1" zeigt am Anfang auch auf das Element. Zur Ausgabe der Liste wird nun in einer Schleife „zei1" jeweils mit dem Nachfolger belegt, solange, bis der Nachfolger den Wert NIL annimmt, also das Ende der Liste erreicht ist. In *Abb. 1.6.2-2* ist die Situation, die nach der Ausgabe besteht, angegeben. Der Wert von „zei1" ist dabei NIL, da dies die letzte Zuweisung im Ausgabeprogramm war. Der Zeiger „basis" zeigt auf den Anfang (hier der letzte Eintrag) der Liste. Wird „basis" auf einen anderen Wert gesetzt, so muß darauf geachtet werden, daß immer irgend ein Bezug auf den Anfang existiert, da sonst die Gefahr besteht, daß ein Element nicht mehr erreicht werden kann.

Abb. 1.6.2-3 zeigt ein weiteres Beispiel mit Pointern. Es handelt sich dabei um ein Trickprogramm, das nicht angewendet werden sollte und auch nicht an allen Rechnern gleich arbeitet. Es ergeben sich auch unterschiedliche Werte. Es soll die absolute Adresse eines Objekts für Testzwecke berechnet werden. Mit einer Variantstruktur ist dies möglich. Dabei wird einmal ein Pointer definiert, zum anderen ein INTEGER-Typ. Wird dem Pointer ein Wert zugewiesen, so kann durch die Variantstruktur der dazugehörige INTEGER-Wert ausgegeben werden. Das Beispiel arbeitet aber wie schon gesagt nicht auf allen Rechnern.

1.6.3 Baumstrukturen

Eine andere Möglichkeit Daten zu organisieren ist, sie in einer Baumstruktur anzulegen. Eine solche Struktur kann mit der folgenden Definition aufgebaut werden:

```
TYPE
  zeiger = ↑baumstruktur;
  baumstruktur =
    RECORD
      links,rechts : zeiger;
      daten : datentyp
    END;
```

Mit den Variablen „links" und „rechts" wird der Anschluß an weitere Objekte vorgenommen. Dabei dürfen zur Erzielung einer Baumstruktur die Pointer „links" und „rechts" immer nur auf neue Objekte zeigen (oder den Wert NIL besitzen). Tun sie das nicht, indem sie z. B. auf den Anfang zeigen, so ergibt sich eine ring- und oder Netzstruktur. *Abb. 1.6.3-1* zeigt ein Beispielprogramm für die Anwendung von Baumstrukturen. Das Beispiel ist an ein Beispiel in [1] angelehnt. Aufgabe ist es, Wörter einzugeben und dann festzustellen, wie oft jedes Wort eingegeben wurde und das Ergebnis ist alphabetisch sortiert auszugeben. *Abb. 1.6.3-2* zeigt eine Beispielseingabe. Das Programm kann dazu verwendet werden, die Häufigkeit von Wörtern in Texten festzustellen. *Abb. 1.6.3-3* zeigt den Aufbau eines Baumes nach der Eingabe der Wörter: EINTRAG MIT BAEUMEN ANZAHL MIT ZAEHLER GEHT. Eine Eingabesequenz wird mit „." abgeschlossen. Bei der Eingabe der Wörter geschieht folgendes: Zunächst wird das Objekt mit dem Wort EINTRAG erzeugt. Beim nächsten Wort wird festgestellt, ob es alphabetisch hinter „EINTRAG" gehört und damit wird ein neues Objekt erzeugt und der Zeiger „rechts" von Eintrag auf das neue Objekt mit „MIT" eingestellt. Wenn ein Objekt schon mit dem

```
PROGRAM baeume(INPUT,OUTPUT); (nach Grogono [1] )

   TYPE
      worttyp = PACKED ARRAY[1..20] OF CHAR;
      zeiger = ↑structur;
      structur =
         RECORD
            links,rechts : zeiger;
            wort : worttyp;
            anzahl : INTEGER
         END;

   VAR
      wortbaum : zeiger;
      neuwort : worttyp;

PROCEDURE lesein(VAR einwort : worttyp);
   VAR
      i : INTEGER;
      ch : CHAR;
   BEGIN
      FOR i:=1 TO 20 DO einwort[i] := ' ';
      i := 1;
      WRITE('-:');
      READ(ch);
      REPEAT
         einwort[i] := ch;
         i := i + 1;
         READ(ch);
      UNTIL EOLN OR (ch=' ') OR (i=21)
   END;

PROCEDURE gibaus(einwort : worttyp);
   BEGIN
      WRITE(einwort)
   END;

PROCEDURE eintrage(VAR baum : zeiger; eintrag : worttyp);
   BEGIN
      IF baum = NIL
         THEN
            BEGIN
               NEW(baum);
               WITH baum↑ DO
```

Abb. 1.6.3-1 Sortieren mit Baumstrukturen

```
                  BEGIN
                     wort := eintrag;
                     anzahl := 1;
                     links := NIL;
                     rechts := NIL
                  END
              END
            ELSE
              WITH baum↑ DO
                IF eintrag < wort
                  THEN eintrage(links,eintrag)
                  ELSE IF eintrag > wort
                         THEN eintrage(rechts,eintrag)
                         ELSE anzahl := anzahl + 1
  END;

PROCEDURE druckbaum(baum : zeiger);
  BEGIN
    IF baum <> NIL
      THEN
        WITH baum↑ DO


          BEGIN
            druckbaum(links);
            gibaus(wort);
            WRITELN(' : ',anzahl);
            druckbaum(rechts)
          END
  END;

BEGIN
  WRITELN;
  wortbaum := NIL;
  REPEAT
    lesein(neuwort);
    WRITELN;
    IF neuwort <> '.                    '
      THEN eintrage(wortbaum,neuwort)
  UNTIL neuwort = '.                    ';
  druckbaum(wortbaum)
END.
```

zu Abb. 1.6.3-1

```
Sorcim PASCAL/M ver 03.05/03.05
-:analyse
-:von
-:woertern
-:in
-:saetzen
-:mit
-:diesem
-:programm
-:kann
-:man
-:sehen
-:ob
-:immer
-:manche
-:woerter
-:in
-:saetzen
-:haeufig
-:verwendet
-:werden
-:und
-:mit
-:welcher
-:anzahl
-:dies
-:vorkommt
-:dies
-:geschieht
-:mit
-:einem
-:programm
-:das
-:einen
-:binaeren
-:baum
-:verwendet
-:.
analyse                  : 1
anzahl                   : 1
baum                     : 1
binaeren                 : 1
das                      : 1
dies                     : 2
diesem                   : 1
einem                    : 1
einen                    : 1
geschieht                : 1
haeufig                  : 1
immer                    : 1
in                       : 2
kann                     : 1
man                      : 1
manche                   : 1
mit                      : 3
ob                       : 1
programm                 : 2
saetzen                  : 2
sehen                    : 1
und                      : 1
verwendet                : 2
von                      : 1
vorkommt                 : 1
welcher                  : 1
werden                   : 1
woerter                  : 1
woertern                 : 1
```

Abb. 1.6.3-2 Beispiel Sortieren von Wörtern

Eintrag existiert, so wird der Anzahl-Zähler um eins erhöht. Dies geschieht nun, bis alle Worte eingegeben sind. Das Ausgabeprogramm arbeitet rekursiv. Es besteht aus der Sequenz:

```
PROCEDURE ausgabe (baumteil: zeiger);
   falls Baumteil nicht leer
      Ausgabe linker Baumteil
      Ausgabe Eintrag und Anzahl
      Ausgabe rechter Baumteil
```

Der Aufruf „ausgabe(wortbaum)" bewirkt also folgendes: Es wird ausgabe (wortbaum.links) aufgerufen. Da dieser Pointer nicht leer ist, wird dann ausgabe(wortbaum.links.links) aufgerufen. Wieder ist er nicht leer. Aber „wort-

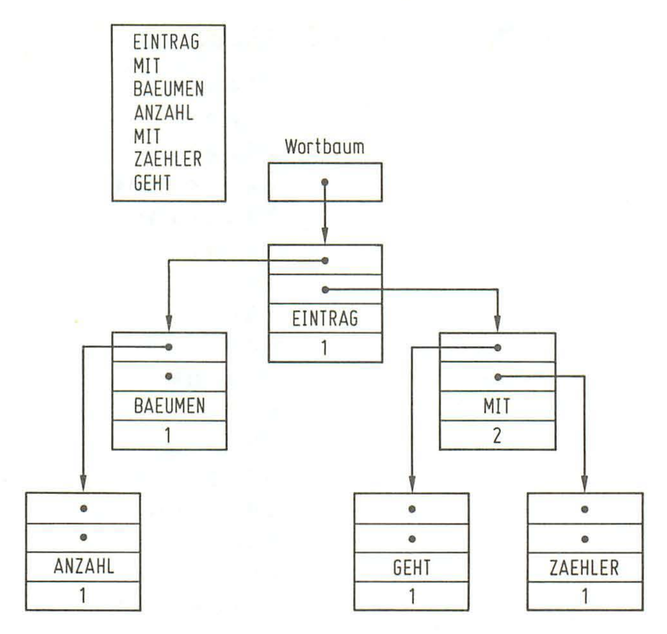


Abb. 1.6.3-3 Aufbau des Baums beim Programmlauf

baum.links.links.links" ist NIL und dann wird nach Rückkehr an die aufrufende Prozedur „Anzahl" und 1 ausgegeben. Der rechte Teil ist auch leer, somit wird nun „BAEUMEN" und 1 angezeigt. Der rechte Teil ist NIL und damit folgt „EINTRAG" und 1. Nun geht es rechts weiter. Beim Objekt „MIT" wird der linke Pointer verwendet, dann gibt es einen NIL Eintrag und „GEHT" und 1 wird ausgegeben, dann wird „MIT" und 2 ausgedruckt und schließlich „ZAEHLER" und 1. Die Ausgabe ist dann beendet.

Das Suchen von bestimmten Wörtern geht bei dieser Struktur schneller als bei einer Listen-Struktur, bei der sequenziell gesucht werden müßte. Eine Listen-Struktur kann hier als Sonderfall aber auch entstehen, wenn die Wörter z. B. alphabetisch sortiert eingegeben werden. Das Programm ist also insbesondere für unsortierte Daten gut geeignet.

1.6.4 Netzstrukturen

Hier soll gezeigt werden, daß für bestimmte Anwendungen auch Netzstrukturen sinnvoll sind. Es sei folgende Aufgabenstellung gegeben: Es soll ein Labyrinth simuliert werden. Das Labyrinth bestehe aus einzelnen Räumen, die über Gänge miteinander verbunden sind. Von einem Raum kann in Nord-, Süd-, Ost- und Westrichtung, falls ein Gang vorhanden ist, weitergegangen werden. Es sollen auch Gänge existieren, die nur in einer Richtung begangen werden können. Es wird zur Realisierung eine Struktur mit vier Zeigern verwendet. Die Zeiger sind mit „sued", „nord", „ost" und „west" benannt. Zeigt ein Zeiger auf sein eigenes Objekt, so gibt es keinen Gang in dieser Richtung.

Abb. 1.6.4-1 zeigt das gesamte Programm. Das Programm kann z. B. zur Simulation einer Umgebung für ein kybernetisches Modell, einer Maus o. ä., ver-

```
PROGRAM labyrinth(INPUT,OUTPUT);

   TYPE
      beschr = PACKED ARRAY[1..20] OF CHAR;
      zeiger = ↑zimmer;
      zimmer =
         RECORD
            sued,nord,ost,west : zeiger;
            raum : beschr
         END;

   VAR
      start,aktuell : zeiger;
      ch : CHAR;

PROCEDURE init;
   VAR
      h1,h2 : zeiger;
   BEGIN
      NEW(aktuell);
      aktuell↑.raum := 'Ausgang             ';
      aktuell↑.sued := aktuell;
      aktuell↑.nord := aktuell;
      aktuell↑.ost := aktuell;
      start := aktuell;
      NEW(aktuell);
      aktuell↑.raum := 'Raum 9              ';
      aktuell↑.ost := start;
      start↑.west := aktuell;
      aktuell↑.west := aktuell;
      aktuell↑.sued := aktuell;
      h1 := aktuell;
      NEW(aktuell);
      aktuell↑.raum := 'Raum 8              ';
      aktuell↑.ost := aktuell;
      aktuell↑.west := aktuell;
      aktuell↑.sued := h1;
      h1↑.nord := aktuell;
      h1 := aktuell;
      NEW(aktuell);
      aktuell↑.raum := 'Raum 6              ';
      aktuell↑.nord := aktuell;
      aktuell↑.sued := h1;
      aktuell↑.west := aktuell;
```

Abb. 1.6.4-1
Labyrinth-Programm

```
    h1 := aktuell; { konserve raum 6 adresse }
    NEW(aktuell);
    aktuell↑.raum := 'Raum 7                  ';
    aktuell↑.sued := aktuell;
    aktuell↑.ost := aktuell;
    aktuell↑.west := aktuell;
    h1↑.ost := aktuell;
    h2 := aktuell; { konserve raum 7 adresse }
    NEW(aktuell);
    aktuell↑.raum := 'Raum 4                  ';
    aktuell↑.sued := aktuell;
    aktuell↑.nord := aktuell;
    aktuell↑.ost := aktuell;
    h2↑.nord := aktuell;
    h2 := aktuell; { raum 4 }
    NEW(aktuell);
    aktuell↑.raum := 'Raum 3                  ';
    aktuell↑.nord := aktuell;
    aktuell↑.sued := aktuell;
    aktuell↑.ost := h2;
    h2↑.west := aktuell;
    h2 := aktuell; { raum 3 }
    NEW(aktuell);
    aktuell↑.raum := 'Raum 2                  ';
    aktuell↑.nord := aktuell;
    aktuell↑.ost := h2;
    h2↑.west := aktuell;
    h2 := aktuell; { Raum 2 }
    NEW(aktuell);
    aktuell↑.raum := 'Raum 5                  ';
    aktuell↑.west := aktuell;
    aktuell↑.sued := aktuell;
    aktuell↑.ost := h1; {raum 6}
    aktuell↑.nord := h2;
    h2↑.sued := aktuell;
    NEW(aktuell);
    aktuell↑.raum := 'Raum 1                  ';
    aktuell↑.nord := aktuell;
    aktuell↑.sued := aktuell;
    aktuell↑.ost := h2;
    h2↑.west := aktuell;
    h2 := aktuell; {raum 1}
    NEW(aktuell);
    aktuell↑.raum := 'Eingang                 ';
    aktuell↑.nord := aktuell;
    aktuell↑.sued := aktuell;
    aktuell↑.west := aktuell;
    aktuell↑.ost := h2;
    h2↑.west := aktuell;
    start := aktuell
END;
```

zu Abb. 1.6.4-1

```
PROCEDURE gehe(dir : CHAR);
   BEGIN
      CASE dir OF
         'n' : aktuell := aktuell↑.nord;
         's' : aktuell := aktuell↑.sued;
         'o' : aktuell := aktuell↑.ost;
         'w' : aktuell := aktuell↑.west
      END
   END;


BEGIN
   WRITELN;
   WRITELN('Sie befinden sich am Anfang eines Labyrinths');
   WRITELN('Sie muessen hineingehen und den Ausgang finden');
   WRITELN('Die Richtungen n,s,o,w sind moeglich');
   WRITELN('Nicht immer kann von einem Raum auch wieder');
   WRITELN('ueber die entgegengesetzte Richtung in den');
   WRITELN('vorhergehenden zurueckgelangt werden');
   init; (besetzten der Raum Definitionen)
   WRITELN(aktuell↑.raum);
   REPEAT
      WRITE('Richtung -:');
      READ(ch);
      WRITELN;
      IF ch IN ['n','s','o','w']
         THEN
            BEGIN
               gehe(ch);
               WRITELN(aktuell↑.raum)
            END
   UNTIL ch = '.'
END.
```

zu Abb. 1.6.4-1

wendet werden. Hier wurde es so ausgelegt, daß man sich selbst in diesem Labyrinth bewegen kann. Es gibt zwei besondere Räume. Einmal ist dies der Raum mit dem Namen „Eingang". Dort beginnt das Labyrinth. Dann gibt es noch einen „Ausgang", den es zu finden gilt. Die Räume im inneren des Labyrinth sind von „Raum 1" bis „Raum 9" durchnumeriert. Als Eingabe kann ein Buchstabe „s", „n", „o" oder „w" eingegeben werden, der als Anfangsbuchstabe der Richtung steht. *Abb. 1.6.4-2* zeigt einen Programmlauf. Die Struktur des Labyrinths ist in *Abb. 1.6.4-3* dargestellt. Die eingezeichneten „Türen" verdeutlichen Stellen, an denen nur in einer Richtung zum nächsten Raum gewechselt werden kann. *Abb. 1.6.4-4* zeigt den Aufbau der Objekte „zimmer". Sie besteht aus vier Zeigern, die die möglichen Verbindungen angeben und einen Namen. In *Abb. 1.6.4-5* ist die Struktur des Labyrinths mit der Darstellung durch Zeiger abgebildet. Existieren zwei Pfeile mit entgegengesetzten Richtungen zwischen zwei Räumen, so kann der Gang zwischen ihnen in beiden Richtungen begangen werden. Pfeile, die auf dasselbe Objekt zeigen geben an, daß kein Gang in dieser Richtung existiert.

```
Sorcim PASCAL/M ver 03.05/03.05
Sie befinden sich am Anfang eines Labyrinths
Sie muessen hineingehen und den Ausgang finden
Die Richtungen n,s,o,w sind moeglich
Nicht immer kann von einem Raum auch wieder
ueber die entgegengesetzte Richtung in den
vorhergehenden zurueckgelangt werden
Eingang
Richtung -:n
Eingang
Richtung -:w
Eingang
Richtung -:o
Raum 1
Richtung -:o
Raum 2
Richtung -:n
Raum 2
Richtung -:s
Raum 5
Richtung -:o
Raum 6
Richtung -:w
Raum 6
Richtung -:n
Raum 6
Richtung -:o
Raum 7
Richtung -:s
Raum 7
Richtung -:w
Raum 7
Richtung -:n
Raum 4
Richtung -:s
Raum 4
Richtung -:w
Raum 3
Richtung -:s
Raum 3
Richtung -:w
Raum 2
Richtung -:s
Raum 5
Richtung -:s
Raum 5
Richtung -:o
Raum 6
Richtung -:s
Raum 8
Richtung -:s
Raum 9
Richtung -:s
Raum 9
Richtung -:w
Raum 9
Richtung -:o
Ausgang
Richtung -:.
```

Abb. 1.6.4-2
Beispiellauf durch das Labyrinth

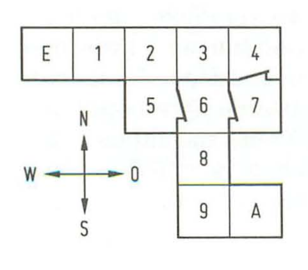


Abb. 1.6.4-3
Aufbau des Labyrinths



Abb. 1.6.4-4
Pointer-Struktur

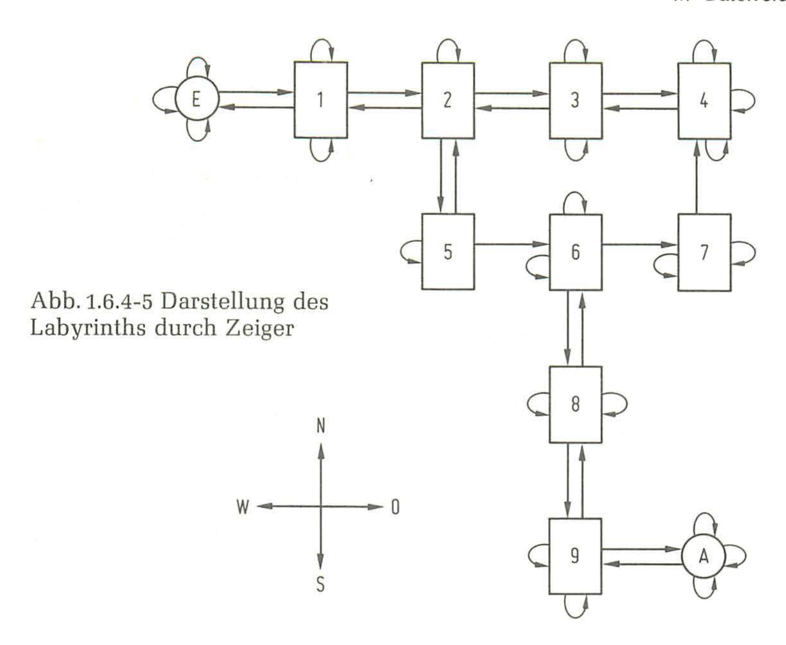


Abb. 1.6.4-5 Darstellung des Labyrinths durch Zeiger

1.7 Dateiverarbeitung

Außer von der Konsole konnten bisher keine Daten bezogen werden. Die Ausgabe war ebenfalls auf Bildschirm oder Drucker beschränkt. Nach Ablauf eines Programms war von diesem nichts hinterlassen worden. Wir wollen jetzt die Dateiverarbeitung kennenlernen. Ein PASCAL-Programm sieht eine Datei wie eine Variable an, deren Speicherplatz aber den des Programms um ein Vielfaches übersteigen kann. Daher kann von einem PASCAL-Programm aus, eine Datei, nur in Teilabschnitten erreicht werden. Eine Datei sieht damit ähnlich aus wie ein Feld von einer Rekord-Struktur, also wie ARRAY [..] OF RECORD... Allerdings kann auf eine Datei nicht über einen Index zugegriffen, sondern nur ein Element nach dem anderen erreicht werden; beginnend mit dem ersten Element. Man sagt auch, die Datei ist für sequenzielle Verarbeitung ausgelegt. Das Gegenteil davon ist ein Random-Zugriff, bei dem eine Art Index angegeben werden kann, was aber im Standard-PASCAL nicht möglich ist.

1.7.1 Definition eines Dateizugriffs

Eine Datei wird z. B. bei der TYPE-Definition angegeben. Es soll zum Beispiel eine Datei aufgebaut werden, die aus Textzeilen mit einer Länge von 80 Zeichen besteht, beispielsweise um einen Texteditor aufzubauen.
Die Definition

```
TYPE
   zeile = PACKED ARRAY [1..80]
   OF CHAR;
   dateityp = FILE OF zeile;
VAR
   datei : dateityp;
```

legt die Datei „datei“ als Variable an. Um sie dem Compiler als Schnittstelle nach außen bekannt zu geben, muß sie ähnlich wie bei Prozeduren Parameter angeben. Auch in der Überschrift erscheinen, also:



Abb. 1.7.1-1 Aufbau einer Datei

PROGRAM dateiverarbeitung
(INPUT,OUTPUT,datei);
INPUT und OUTPUT sind die Standard-IOs, die bei READ und WRITE verwendet werden und deshalb auch in der Überschrift erscheinen müssen. Hier gibt es bei den Mikrorechnern-Implementationen Unterschiede, die im jeweiligen Handbuch nachgesehen werden müssen. Eine Datei sieht im Prinzip wie in *Abb. 1.7.-1* aus. Hier sind z. B. fünf Komponenten sichtbar.
Mit
datei↑
kann auf eine solche Komponente zugegriffen werden, wenn die Datei mit der obigen Definition übereinstimmt.

1.7.2 Schreiben auf Dateien

Eine Datei ist zunächst einmal leer, wenn sie nicht schon vorher existierte und es müssen Daten hineingeschrieben werden. Dabei geschieht dies ebenfalls komponentenweise. *Abb. 1.7.2-1* zeigt den Ablauf. Im Fall a) ist die Datei zunächst noch leer, bei b) wurde eine Komponente geschrieben und bei c) noch eine weitere. Um so vorzugehen, wird zunächst der Befehl REWRITE(datei) angegeben. Der Datei-Zeiger wird

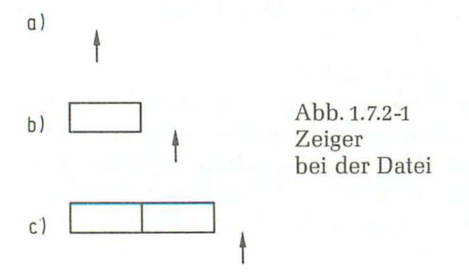


Abb. 1.7.2-1 Zeiger bei der Datei

damit auf den Anfang der Datei gelegt und eine etwaige Information, die vorhanden war, wird damit gelöscht. Mit PUT(datei↑) wird eine Komponente in die Datei geschrieben, und der Datei-Zeiger wandert an die nächste freie Stelle. Nun muß aber zuvor die Variable „datei↑" mit Daten belegt werden. Die kann z. B. wie folgt durchgeführt werden:

Mit
VAR
buffer : zeile;
wird ein Feld definiert, daß genauso wie eine Komponente der Datei 80 Zeichen beinhalten kann. Dieses Feld kann nun mit irgendwelchen Daten belegt werden. Dann wird mit
datei↑ := buffer;
der Inhalt des Feldes an die Datei-Variable zugewiesen. Mit
PUT(datei↑);
wird die Information in der Datei abgelegt. Die Funktion WRITE kann ebenfalls verwendet werden. Es gilt dabei:
datei↑ := buffer;
PUT(datei↑);
kann zu
WRITE(datei,buffer);
vereinfacht werden. WRITELN kann auch verwendet werden, wenn es sich beim Dateityp um eine Text-File handelt.

TYPE
TEXT = PACKED FILE OF CHAR;
Dabei ist TEXT ein Standard-Typ, der nicht definiert werden muß und mit

VAR
textdatei : TEXT;
wird eine Datenvariable definiert, deren Datei aus Einzelzeichen besteht.

Die Funktion
WRITE(datei,komp1,komp2,komp3);
wirkt wie drei einzelne WRITE-Anweisungen.

1.7.3 Lesen aus einer Datei

Beim Lesen wird nun umgekehrt verfahren. Mit RESET(datei) wird der Datei-Zeiger auf den Anfang der Datei gestellt. Es wird dann sofort die erste Komponente in die Variable datei↑ gestellt. Mit dem Befehl GET(datei↑) wird die nächste Komponente gelesen. In *Abb. 1.7.3-1* ist die Situation dargestellt. Nach dem RESET-Befehl wird die erste Komponente eingelesen (a). Mit dem GET-Befehl wird die Situation b) erreicht. Ein weiterer GET-Befehl liest die dritte Komponente ein und der Fall c) liegt vor. Wird auf eine Komponente zugegriffen, die nicht existiert, so kann mit der Booleschen Funktion EOF(datei) der Tatbestand festgestellt werden. Sie nimmt den Wert TRUE an, wenn auf eine Komponente zugegriffen wird, die nicht definiert ist.

Das Lesen aus einer Datei kann also beispielsweise wie folgt geschehen:

```
RESET(datei);
buffer := datei↑;
  (· erste Komponente ·)
GET(datei↑);
buffer := datei↑;
  (· zweite Komponente ·)
...
```

Nach jeder Eingabe gibt es zwei Möglichkeiten:

EOF(datei) ist FALSE, dann enthält datei↑ gültige Daten.

EOF(datei) ist TRUE, dann ist datei↑ undefiniert und das Dateiende erreicht.

Anstelle von GET kann auch READ verwendet werden. Dabei gilt:

```
buffer := datei↑;
GET(datei↑);
```

ist gleichwertig mit

```
READ(datei, buffer);
```

Bei Textdateien gibt es auch die Möglichkeit mit READLN einzulesen und dabei Daten bis zum Zeilenende zu

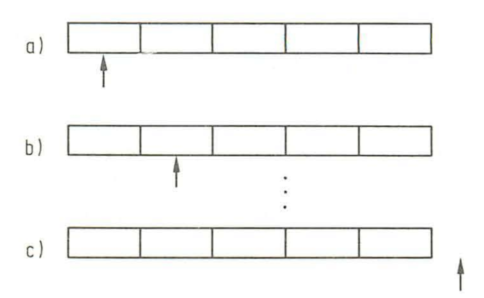


Abb. 1.7.3-1 Zeiger bei der Datei

überlesen. Mit der Funktion EOLN(textdatei) kann ein Zeilenende auch per Programm abgefragt werden.

```
READLN(textdatei)
```

ist gleichwertig zu

```
WHILE NOT EOLN(textdatei) DO
   GET(textdatei↑);
 GET(textdatei↑);
```

wobei für „textdatei" die Definition

```
VAR
  textdatei : TEXT;
```

gilt.

Mit READ(datei,buf1,buf2,...bufn) wird eine Sequenz von READ(datei,bufi) abgekürzt.

1.8 Sonderfunktionen

1.8.1 Die GOTO-Anweisung

Viele, die schon mit einer anderen Programmiersprache, wie BASIC und FORTRAN gearbeitet haben, werden sich wundern, daß wir bisher nicht ein einziges Mal eine Sprunganweisung verwendet haben. Für die Programmierung von beliebigen Problemen ist dies tatsächlich in PASCAL nicht nötig, da sich alle Probleme mit strukturierten Anweisungen, wie WHILE, IF THEN ELSE ect., lösen lassen. In PASCAL gibt es dennoch eine GOTO-Anweisung um z. B. für Fehlerfälle einen Sprung auf eine Fehlerbehandlung zu ermöglichen, ohne die Pro-

grammstruktur durch IF-Anweisungen zu sehr undurchsichtig zu machen.

Die Sprungmarken müssen definiert werden. Dies geschieht mit der LABEL-Anweisung. Sie wird vor der CONST, TYPE...Definition angegeben. Die Marken können nur ganze Zahlen sein.
Beispiel:

```
LABEL 1,2;
```

Ein Sprung kann dann lauten:

```
        GOTO 1;
oder    GOTO 2;
```

die Einsprungstelle wird mit

```
        1: anweisung;
oder    2: anweisung;
```

gekennzeichnet.

1.8.2 Prozeduren und Funktionen als Parameter

Eine, leider nur bei wenigen Compilern vorhandene Möglichkeit, Prozeduren und Funktionen als Parameter zu übergeben, bietet einen sehr interessanten Aspekt.
Es könnte dann zum Beispiel eine Prozedur

```
PROCEDURE kurvendiskusion
    (FUNCTION f : REAL);
```

definiert werden. Sie soll eine Kurvendiskusion ausführen. Es ist dann möglich z. B. falls

```
FUNCTION parabel(X:REAL):REAL;
  BEGIN
    parabel := X·X
  END;
```

definiert ist mit

```
kurvendiskusion(parabel);
```

die Funktion „parabel“ als Parameter an „kurvendiskusion“ zu übergeben. In der Prozedur „kurvendiskusion“ wird mit f(X) allgemein gearbeitet.

2 Mikrorechner PASCAL-Realisierungen

Wir hatten uns bisher auf das Standard-PASCAL beschränkt, so wie es von Jensen & Wirth in [2] definiert wurde. In den folgenden Abschnitten soll nun auf verschiedene PASCAL-Systeme eingegangen werden, wie sie für Mikroprozessoren erhältlich sind. Dabei ergeben sich Erweiterungen des Sprachumfangs und leider auch ein paar Einschränkungen, je nach PASCAL-Version. Ein Problem bei den Mikrorechnern ist der beschränkte Speicherplatz. Da PASCAL eine recht umfangreiche Sprache ist, wird sehr viel Platz für die Compiler benötigt. Die PASCAL-Systeme liegen daher meist knapp unter der maximalen Speichergrenze (48K Bytes bis 56K Bytes von max. 64 K Bytes). Die Quellen werden mit einem Editor erstellt und sind auf einer Diskette oder Platte abgelegt. Der Compiler übersetzt sie entweder in einen Zwischencode (bei PASCAL P-Code genannt) oder in Maschinensprache. Das Ergebnis wird wieder auf Diskette abgelegt. Danach kann im Falle der Maschinensprache das Programm direkt gestartet werden, oder es wird mit einem P-Code-Interpreter ausgeführt. Wie eine Mikrorechnerkonfiguration aufgebaut sein kann, ist ausführlich in den Büchern 3, 4 dargestellt (siehe Anhang).

2.1 Das UCSD-PASCAL

Das wohl verbreitetste PASCAL-System ist das UCSD-PASCAL , das von der University of California, San Diego stammt [8,9]. Der PASCAL-Compiler erzeugt einen P-Code, der dann interpretiert wird. Zum UCSD-PASCAL gehört aber nicht nur der Compiler, sondern auch ein Editor, sowie ein Betriebssystem. Da die gesamte Systemsoftware selbst in PASCAL geschrieben ist, und in P-Code-Form vorliegt, ist es möglich gewesen, das UCSD-System unabhängig vom Mikrorechner zu verwenden. Es wird nur der P-Code-Interpreter neu geschrieben, sowie der Code-Generator eines ebenfalls verfügbaren Assemblers. Damit ist das gesamte System auf einem beliebigen Mikrorechner verfügbar. Das UCSD-System existiert damit schon für die Rechner 8080,Z80,6800,6809,6502 usw. Da auch das Betriebssystem einheitlich ist, können Programme von verschiedenen Mikrorechnern, die in PASCAL für dieses System geschrieben wurden, untereinander ausgetauscht werden.

Das UCSD-PASCAL bietet einige Erweiterungen an, die durch die große Verbreitung fast schon ein neuer Standard geworden sind und für das Schreiben von PASCAL-Programmen sehr nützlich sind. Die neuen Befehle werden daher in den folgenden Kapiteln näher beschrieben.

2.1.1 String-Verarbeitung

In PASCAL sind nur Definitionen der Art PACKED ARRAY .. OF CHAR Für Zeichenketten möglich. Hier wird ein neuer Typ STRING eingeführt, der es

erlaubt, Strings mit unterschiedlicher Länge zu verarbeiten. Die Definition

```
VAR
   name : STRING;
```

deklariert „name" als einen String, der maximal 80 Zeichen lang sein kann. Soll eine andere Länge reserviert werden, so ist die Größe in Klammern anzugeben:

```
VAR
   zeichenkette : STRING[16];
```

In „zeichenkette" können maximal 16 Zeichen stehen. Die Betonung liegt dabei auf maximal, da auch weniger Zeichen enthalten sein können. Strings mit unterschiedlichen Längen können verglichen werden, wobei lexiografische Ordnung zugrunde liegt.

Für Strings gibt es eine Reihe von nützlichen Funktionen:

```
FUNCTION LENGTH
( zeichenkette : STRING ) : INTEGER;
```

Die aktuelle Länge eines Strings kann dadurch ermittelt werden.

LENGTH('1234') ergibt 4

und

LENGTH('') ergibt 0

```
FUNCTION POS(such : STRING,
quell : STRING) : INTEGER;
```

Damit läßt sich die erste Position von „such" in „quell" ermitteln. Beispiel:

```
suche := 'FIND';
suchin := 'ICH FINDE';
WRITELN(POS(suche,suchin));
```

ergibt: 5, wobei „suche" und „suchin" vom Typ STRING sind.

```
FUNCTION CONCAT
(quelltexte, ... : STRING) : STRING;
```

Mehrere durch Kommatas getrennte Strings können zu einem neuen verbunden werden:

```
anfang :='START';
ende := 'STOP';
WRITELN(CONCAT
(anfang,' MITTE ',ende));
```

gibt den Ausdruck:

START MITTE STOP

```
FUNCTION COPY(quell : STRING;
startpos, anzahl : INTEGER) :
STRING;
```

Es wird ein String mit „anzahl" Zeichen von der Position „startpos" aus als Ergebnis übergeben:

```
daten := 'TEXT MIT ZEICHEN';
WRITELN(COPY(daten,6,3));
```

ergibt:

MIT

als Ausdruck.

```
PROCEDURE DELETE
(ziel : STRING; startpos,
anzahl : INTEGER);
```

Beginnend mit „startpos" werden „anzahl" Zeichen aus dem String „ziel" entfernt.

```
zeichen := 'TEXT UND AUSGABE';
DELETE(zeichen, 6,4);
WRITELN(zeichen);
```

ergibt:

TEXT AUSGABE

```
PROCEDURE INSERT
(quelle, ziel : STRING;
position : INTEGER);
```

Die Zeichen in „quelle" werden nach „ziel" beginnend bei „position" eingefügt.

Beispiel:

```
einfuegen := 'EINGEFUEGTER';
zieltext := 'DIES IST EIN TEXT';
INSERT(einfuegen,zieltext,
POS('TE',zieltext));
```

in „einfuegen" steht dann:

DIES IST EIN EINGEFUEGTER TEXT

Im UCSD-PASCAL ist es möglich, einen INTEGER mit hoher Genauigkeit zu definieren Dies geschieht durch Angabe der gewünschten Dezimalstellen in Klammer.

Beispiel:

```
VAR X : INTEGER[8];
```

definiert eine Variable X mit acht Dezimalen, die auch ein Vorzeichen haben kann. Der Index darf nicht größer als 36 sein.

PROCEDURE STR(longinteger : long;
zielstring : STRING);

Die INTEGER-Variable „longinteger" wird in einen String konvertiert. Beispiel für die Anwendung:

```
langint := 1234567891;
STR(langint,stringvar);
INSERT('.',stringvar,
        PRED(LENGTH(stringvar)));
WRITELN(stringvar);
```

ergibt

12345678.91

2.1.2 Verarbeitung mit Zeichenfeldern

Nicht nur für den neuen Typ STRING gibt es neue Befehle, sondern auch für den Typ PACKED ARRAY .. OF CHAR sind Erweiterungen vorgesehen.

FUNCTION SCAN
(laenge : INTEGER;
ausdruck : teilausdruck;
zeichenfeld : PACKED ARRAY .. OF CHAR) : INTEGER;

Damit läßt sich eine Anzahl von Zeichen bestimmen, die entweder die in „laenge" angegebene Maximalzahl darstellt, oder die Position, die durch „ausdruck" bestimmt wird, beginnend mit 0. Ist die „laenge" negativ, so wird rückwärts gesucht. Bei „zeichenfeld" kann ein Startindex angegeben werden.

Als Teilausdruck gilt:

„⟨⟩" oder „=" gefolgt von einem Zeichenausdruck.

Beispiel:

test := 'ABCDEFABCDEF';
SCAN(100,⟨⟩'A',test) ergibt den Wert 1.

PROCEDURE MOVELEFT (quellfeld, zielfeld : PACKED ARRAY .. OF CHAR; laenge : INTEGER);
PROCEDURE MOVERIGTH
(... wie oben ...);

Mit diesen beiden Prozeduren kann ein Blocktransport ausgeführt werden. Dabei beginnt MOVELEFT auf der linken Seite und transportiert nach der linken Seite des Ziels und MOVERIGHT transportiert beginnend mit der rechten Seite. Es werden „laenge" Bytes transportiert. Es kann durch Indizierung der Felder ein Startpunkt gegeben werden.

PROCEDURE FILLCHAR
(zielfeld : PACKED ARRAY .. OF CHAR; laenge : INTEGER;
zeichen : CHAR);

Ein „zielfeld" wird mit „laenge" Zeichen aufgefüllt. Die Anweisung ist gleichwertig zu:

```
a[0] := zeichen;
MOVELEFT
(a[0],a[1],laenge-1);
```

FILLCHAR ist allerdings viel schneller.

2.1.3 IO-Verarbeitung

PROCEDURE RESET (datei : FILE);
PROCEDURE RESET
(datei : FILE; dateiname : zeichenkette);
PROCEDURE REWRITE
(datei : FILE; dateiname : zeichenkette);

Hier wird von der Standard-Definition etwas abgewichen. Auf der Diskette sind Dateien abgelegt, die einen Namen besitzen. Nun muß dieser Name dem PASCAL bekannt gegeben werden, entweder um dann aus dieser Datei lesen zu können, oder um sie zu beschreiben. Der „dateiname" ist dabei den UCSD-Konventionen gemäß aufgebaut. Möglich wäre z. B.:

```
RESET(datei,'LIES.MICH');
REWRITE
(ausgebendatei,'PRINTER:');
```

Im ersten Fall wird die Datei „LIES. MICH" zum Lesen geöffnet, im zweiten Fall wird die Ausgabe auf den Drucker ermöglicht. Weitere Voreinstellungen sind:

CONSOLE: ; Bildschirm und Tastatur, die aber voreingestellt ist

KEYBOARD: ; Tastatur ohne Echo
REMIN: ; Remote input
REMOUT: ; Remote output
PRINTER: ; Druckerausgabe
Wird RESET ohne „dateiname“ angegeben, so wird nur der Dateizeiger auf den Anfang zurückgestellt, wobei schon geöffnet worden sein muß.

PROCEDURE UNITREAD(unitzahl : INTEGER; zeichenfeld: PACKED ARRAY; laenge : INTEGER; [diskblockadr : INTEGER; zahl : INTEGER]);
PROCEDURE UNITWRITE(... wie UNITWRITE ...);
Die „unitzahl“ gibt die Geräteadresse an.
#1: CONSOLE:
#2: KEYBOARD:
#4: Drive 1
#5: Drive 2
#6: PRINTER:
#7: REMIN:
#8: REMOUT:
Im „zeichenfeld“, das ein PACKED ARRAY kann auch mit einem Index versehen werden, dieser wird dann als Start-Adresse genommen. In „laenge“ ist die Anzahl der zu übertragenden Bytes anzugeben. Die „diskblockzahl“ wird nur bei Diskettenzugriffen benötigt. Sie stellt die absolute Blockzahl dar, bei der der Transfer beginnt. Wird sie nicht angegeben, so ist 0 voreingestellt. Mit „zahl“ kann, falls 1 angegeben ist (0 ist voreinst.), bestimmt werden, daß die Übertragung asynchron vorgenommen werden soll.
FUNCTION UNITBUSY
(unitzahl : INTEGER) : BOOLEAN;
Falls die Funktion den Wert TRUE erhält, wartet das angesprochene Gerät auf Beendigung eines IO-Zyklus.
PROCEDURE UNITWAIT
(unitzahl : INTEGER);
Es wird solange gewartet, bis das angesprochene Gerät frei ist. Die Anweisung ist identisch mit:
WHILE UNITBUSY(n) DO (warten);
PROCEDURE UNITCLEAR
(unitzahl : INTEGER);
Die IO-Treiber werden in den Urzustand zurückgesetzt.
FUNCTION BLOCKREAD
(datei : FILE;
zeichenfeld : PACKED ARRAY ..;
bloecke : INTEGER;
[reladr : INTEGER]) : INTEGER;
FUNCTION BLOCKWRITE
(... wie oben ...) : INTEGER;
Als Ergebnis wird ein Wert übergeben, der die Anzahl der übertragenen Blöcke darstellt. Die Datei muß als „untyped“ deklariert sein, daß heißt hier als F : FILE; ohne Angabe des Fileaufbaus. Ein Block ist hierbei 512 Bytes lang. Die Länge von „zeichenfeld“ muß ein ganzzahliges Vielfaches der Blocklänge sein. Mit „bloecke“ wird die Anzahl der gewünschten Blöcke angegeben, die übertragen werden sollen. Mit „reladr“ kann die Blocknummer angegeben werden, die beginnend mit 0 vom Datei-Anfang gezählt wird. Wird der Parameter „reladr“ nicht angegeben, so erfolgt ein sequenzieller Zugriff. Mit EOF(datei) kann geprüft werden, wann der letzte Block einer Datei gelesen wurde.
PROCEDURE CLOSE(datei : FILE);
PROCEDURE CLOSE
(datei : FILE; option : optionen);

Als Option gibt es:
LOCK
NORMAL
PURGE
CRUNCH
Wird keine „option“ angegeben, so gilt dasselbe wie für NORMAL. Der Datei-Status wird auf Close gesetzt. Wurde die Datei mit REWRITE eröffnet, so wird sie anschließend gelöscht.

Wird LOCK angegeben, so bleibt bei Diskettenzugriffen die Datei auf der Diskette nach dem CLOSE-Befehl erhalten.

Wird PURGE angegeben, so wird der Dateiname der zu „datei“ gehört, gelöscht.

Bei CRUNCH wird eine „datei“ geschlossen, so daß nur der minimale Bedarf auf der Diskette erhalten bleibt.

FUNCTION IORESULT : INTEGER;

Es wird hier im Fehlerfall eine Fehlernummer angegeben, sonst ist der Wert 0.

PROCEDURE PAGE (datei : FILE);

Ein Form-Feed wird an die Datei übergeben (ASCII FF).

PROCEDURE SEEK
(datei : FILE; zahl : INTEGER);

Damit ist es möglich, auch im Random-Access auf die Datei zugreifen zu können. Es wird der „zahl“te Rekord angesprochen und der Dateizeiger dorthin gesetzt, so daß der nächste Zugriff mit GET, PUT dort ausgeführt wird. Zwischen zwei SEEK-Befehlen muß mindestens ein GET oder PUT ausgeführt werden, da sonst nicht vorhersehbare Ergebnisse folgen.

Die Funktionen EOF, EOLN und die Prozeduren GET, PUT, READ(LN), WRITE(LN) werden standardgemäß ausgeführt, nur, daß bei READ(LN) und WRITE(LN) die Einschränkung gilt, Zugriffe nur bei Dateien des Typs TEXT (FILE OF CHAR) oder INTERACTIVE durchzuführen. INTERACTIVE ist ein neuer Typ, der ein paar interessante Möglichkeiten schafft.
Mit

```
VAR
   dateiinteract : INTERACTIVE;
```

wird eine Datei definiert. Bei einer normalen Datei, die durch FILE deklariert wurde, erfolgt beim Lesen gleich nach dem RESET der erste Zugriff. Bei einer INTERACTIVE-Datei ist dies nicht so. Denn ein Zeichen soll erst dann gelesen werden, wenn ein Aufruf z. B. mit READ erfolgt. Dies ist bei den INPUT- und OUTPUT-Dateien schon immer so. Sie sind auf die Konsole eingestellt und werden immer dann verwendet, wenn keine Datei bei READ und WRITE angegeben wird. Wird ein READ-Befehl von der Konsole ausgeführt, kann erst dann eine Eingabe von dort verlangt werden. Mit dem neuen Datentyp kann dies nun auch allgemein gefordert werden. Die Definition für READ ändert sich damit:

READ(dateiinteract, buffer);

entspricht nun

GET(dateiinteract↑);
buffer := dateiinteract↑;

aber nur beim Typ INTERACTIVE.

2.1.4 Diverse Befehle

FUNCTION SIZEOF
(parameter : Variablen
oder Type-Name) : INTEGER;

Es wird die Anzahl der Bytes übergeben, die der Parameter im Stack belegt. Die Funktion ist z. B. nützlich bei FILLCHAR oder MOVE ... Befehlen.

FUNCTION TIME
(VAR hoeherwertig,
niederwertig : INTEGER);

Die Uhrzeit in 50zigstel einer Sekunde wird übergeben. der Wert ist aber Hardwareabhängig.

FUNCTION LOG
(zahl : REAL) : REAL;

Der Logarithmus zur Basis 10 wird berechnet.

FUNCTION PWROFTEN
(exponent : INTEGER) : REAL;

Als Ergebnis wird 10^{exponent} übergeben. „exponent“ muß im Bereich 0..37 liegen.

PROCEDURE MARK
(VAR heapzeiger : ↑INTEGER);
PROCEDURE RELEASE
(VAR heapzeiger : ↑INTEGER);

Mit diesen beiden Prozeduren kann die Prozedur DISPOSE approximiert werden, da sie nicht verfügbar ist. Werden

dynamische Variablen kreiert, so wird alles auf einem Stack angelegt. Dazu gibt es einen „heapzeiger". Soll wieder Platz auf diesem Stack gemacht werden, so kann nicht einfach irgend ein Element gelöscht werden, sondern immer nur vom Ende des Stack bis zu einer bestimmten Position. Diese Position kann mit MARK festgehalten werden. Dann werden z. B. von dort aus mit NEW neue Elemente angelegt. Wird dann RELEASE ausgeführt, so werden alle von dieser Position aus angelegten Variablen wieder entfernt und der Platz ist für das Anlegen von neuen Variablen wieder frei.

PROCEDURE HALT;

Die Programmausführung wird gestoppt.

PROCEDURE GOTOXY
(xkoord, ykoord : INTEGER);

Der Cursor wird an die Stelle des Bildschirms gebracht, der durch die beiden Koordinaten angegeben wurde. Dabei ist (0,0) links oben.

2.1.5 Weitere Unterschiede zum Standard-PASCAL

In der CASE-Anweisung ist das Ergebnis der Ausführung undefiniert, wenn es keine Marke gibt, die im CASE-Selektor steht. Im UCSD-PASCAL wird dann die Ausführung mit der nächsten Anweisung fortgesetzt, die der CASE-Anweisung folgt. Vor der dem END in Variant-Definitionen ist kein Strichpunkt zugelassen, was im Standard möglich wäre.

EOF(datei)

Als EOF-Zeichen gilt auch CTRL-C.

GOTO und EXIT

GOTO-Anweisungen sind nur innerhalb eines Blocks erlaubt. Mit EXIT ist eine Möglichkeit gegeben, mit einem Parameter, der einen Prozedurnamen angibt, die angegebene Prozedur zu verlassen.

Segment-Prozeduren

Die einzelnen Segmente müssen nicht alle auf einmal im Speicher sein, sie werden vielmehr dynamisch vom Laufzeitsystem geholt. Das UCSD-PASCAL-System ist selbst auf diese Weise aufgebaut, da es sehr groß ist.

Beispiel:

```
PROGRAM segmente;
(· definitionen ·)
PROCEDURE drucke
( t : STRING); FORWARD;
SEGMENT PROCEDURE eins;
   BEGIN PRINT (' seg 1') END;
SEGMENT PROCEDURE zwei;
   BEGIN PRINT (' seg 2') END;
PROCEDURE print;
(· kein parameter da forward ·)
   BEGIN writeln ( t ) END;
BEGIN
   eins; zwei
END.
```

Es ergibt sich:

```
seg 1
seg 2
```

Hier wurde übrigens FORWARD verwendet. Eine Möglichkeit, eine Prozedur oder Funktion zu definieren, bevor sie niedergeschrieben wurde. FORWARD wird angewendet, da SEGMENT-Prozeduren an erster Stelle stehen müssen.

PASCAL-Programme können auch getrennt übersetzt werden und dann zu einem Programm gebunden werden. Dazu werden die reservierten Worte UNIT, INTERFACE,IMPLEMENTATION und USES verwendet. Assembler-Unterprogramme können durch EXTERNAL angegeben werden. Wegen des Umfangs sei hier auf die UCSD-Manuale verwiesen [8,9].

2.2 PASCAL/M

Dieses PASCAL-System wurde bei der Erstellung der Programmbeispiele des Buches verwendet. Das PASCAL läuft auf dem CP/M-Betriebssystem mit einem Z80-Prozessor und mindestens 56K Bytes RAM. Das CP/M-Betriebssystem ist in [3] beschrieben. PASCAL/M [10] ist sehr ähnlich zu dem UCSD-PASCAL, besitzt aber doch einige Abweichungen. Der Compiler erzeugt aber ebenfalls einen P-Code, der dann mit einem P-Code-Interpreter ausgeführt wird.

2.2.1 String-Verarbeitung

Wie auch beim UCSD-PASCAL gibt es den Typ STRING. Die Funktionen

LENGTH	POS	INSERT
DELETE	CONCAT	COPY

haben die gleiche Bedeutung und Definition wie im UCSD-PASCAL.
Neu sind:

```
  PROCEDURE WDECA
  (zahl : INTEGER;
  zeichenkette : STRING);
  PROCEDURE WHEXA
  (zahl : INTEGER;
  zeichenkette : STRING);
```

Die Prozedur WDECA wandelt eine vorzeichenlose Zahl in eine Zeichenkette mit dezimaler Darstellung. Bei WHEXA erfolgt die Umwandlung in das sedezimale (veralt. hexadezimal) Zahlensystem.

2.2.2 Verarbeitung mit Zeichenfeldern

Die Funktionen entsprechen exakt denen des UCSD-PASCALS.

SCAN	MOVELEFT
MOVERIGHT	FILLCHAR

2.2.3 IO-Verarbeitung

Die Prozeduren RESET und REWRITE sind ähnlich zu denen im UCSD-PASCAL, nur das bei den Datei-Angaben die CP/M-Konventionen eingehalten werden müssen.

Beispiel:

```
  RESET(datei,'A:LIES.MIC');
  (· Eröffnen der Datei
  LIES.MIC auf Drive A ·)
REWRITE(datei1,'LP:');
  (· Ausgabe auf den Drucker ·)
```

Das Format des Dateinamens ist:

	name.extn
oder	device:name.extn
oder	device:

Für „device" kann stehen:

A:	Drive 0
B:	Drive 1
	...
H:	Drive 7
CONSOLE:	Benutzerkonsole
CON:	"
CRT:	Konsole ohne Echo
READER:	RDR-Gerät
PUNCH:	PUN-Gerät
PRINTER:	Drucker (LST)
LP:	"
NULL:	Null-Gerät

Alle Geräte verhalten sich wie im BIOS des CP/M-Systems definiert wurde bis auf CRT: und NULL:. Eine Eingabe von CRT: erfolgt ohne Echo, also ohne daß das eingegebene Zeichen auf dem Bildschirm wieder ausgegeben wird. Ein Transfer mit NULL: wirkt wie die Angabe irgend eines anderen Gerätes, nur daß kein Datentransport stattfindet.

```
PROCEDURE PURGE
(dateiname: STRING)
```

Wenn die angegebene Datei existiert, so wird sie gelöscht.

Beispiel:
PURGE('istschon.da');
(· loeschen falls da ·)
REWRITE(datei,'istschon.da');
(· neu anlegen ·)

READ, READLN, WRITE, WRITELN, GET, PUT, PAGE, EOF, EOLN arbeiten wie im UCSD-PASCAL.
PROCEDURE CLOSE(datei:FILE);
PROCEDURE CLOSE
(datei:FILE;LOCK);
PROCEDURE CLOSE
(datei:FILE;PURGE);

Die Datei wird zunächst geschlossen. Der Pointer „datei↑" ist danach undefiniert. Bei den ersten beiden Varianten wird die Datei geschlossen und bleibt auf der Disk erhalten. Bei der letzten Form wird die Datei gelöscht. Wird CLOSE in einem Programm nicht verwendet, so werden die Dateien nach Programmende automatisch geschlossen.

FUNCTION FILEBUSY
(datei:FILE) : BOOLEAN;

Wirkt ähnlich wie UNITBUSY im UCSD-PASCAL. Es muß eine Datei mit CRT: oder CONSOLE: vorliegen. Der Wert der Funktion ist TRUE, wenn keine Daten von der Konsole anliegen.
PROCEDURE FILEREAD(datei:FILE;
feld:PACKED ARRAY OF ..;
länge:INTEGER; [reladr:INTEGER]);
PROCEDURE FILEWRITE(...);
Hier ist ein Direktzugriff auf Dateien des Typs „untyped" möglich, wie im UCSD-PASCAL mit UNITREAD und UNITWRITE. Mit IORESULT kann wie im UCSD-PASCAL eine Fehlernummer geholt werden.

Eine Random-Datei-Verwaltung ist ebenfalls möglich. Die Dateien müssen aber eine „Nicht-Text-Datei" sein.
FUNCTION SIZE(VAR f : FILE OF
irgend_ein_Typ) : INTEGER;
Als Ergebnis wird die Anzahl der in der Datei vorhandenen Elemente eine offene Datei übergeben.
FUNCTION NEXT(VAR f : FILE OF
irgend_ein_Typ) : INTEGER;
Von einer geöffneten Datei kann die Position des Datei-Zeigers als Ergebnis erhalten werden. Der Wert liegt im Bereich (0..SIZE(f)-1) und stellt die Kardinalzahl für den nächsten Datei-Record dar, der mit GET oder PUT angesprochen wird.
PROCEDURE SET_NEXT
(VAR f : FILE OF irgend_ein_Typ;
pos : INTEGER);
Damit kann der Datei-Zeiger auf eine beliebige Stelle von Dateien gesetzt werden. Die Records einer datei werden von 0 bis SIZE(f)-1 durchnumeriert. Wird der Zeiger außerhalb der Datei positioniert, so wird EOF(f) TRUE.

2.2.4 Diverse Befehle

Neben den beiden Kommentarklammern (· ·) und { } gibt es im PASCAL/M auch noch eine andere Möglichkeit, Kommentare zu bilden:
- - Kommentar
Der Kommentar beginnt mit den beiden Zeichen - - und endet mit dem Zeilenwechsel. Dies ist eine Darstellung, wie sie auch in ADA [13] verwendet wird.

Die Funktionen SIZEOF, LOG, PWROFTEN, EXIT, HALT MARK, RELEASE entsprechen denen des UCSD-PASCALS.
FUNCTION MEMAVAIL: INTEGER;
Mit dieser Funktion läßt sich der verbleibende Speicherplatz bei der Ausführung ermitteln.
FUNCTION RANDOM
(seed: INTEGER): REAL;
Als Ergebnis wird ein Wert zwischen 0.00 und 1.00 geliefert. Wenn die Varia-

ble „seed" 0 ist, so wird die nächste Zufallszahl in der Sequenz geliefert. Mit einem anderen Wert startet die Sequenz erneut und die Zahlangabe bestimmt, mit welchem Wert angefangen wird.

PROCEDURE TIME
(uhrzeit, datum : STRING);

In „uhrzeit" steht die aktuelle Zeit in der Form 'HH:MM:SS' und in „datum" 'mm/dd/yy'. Die Werte werden dabei aus Speicherzellen entnommen, die von einem eigenen Programm verwaltet werden müssen:

10H=mm, 11H=dd, 12H=yy,
13H=hh, 14H=mm, 15H=ss.

Ähnlich wie im UCSD-PASCAL gibt es hier auch einen Long-Integer, der allerdings anders definiert wird und auch andere Eigenschaften besitzt. Die Angabe erfolgt mit dem reservierten Wort LONG_INTEGER.

Beispiel:

```
VAR
   lang : LONG_INTEGER;
```

Der größte Wert bei einem Standard-Integer ist MAXINT (32767); bei einem Long-Integer ist er MAXLINT = 2147483647.

Ist ein Long-Integer nicht größer als MAXINT, so kann er direkt an einen Integer zugewiesen werden. Long-Integer können an allen Stellen, wo auch Integer erlaubt sind, verwendet werden. Die Umwandlung zwischen verschiedenen Integer wird automatisch vorgenommen, außer wenn sie aufgrund des Bereichs nicht möglich ist.

2.2.5 Konsol-Verarbeitung

Um einfach mit der Konsole arbeiten zu können, gibt es eine Reihe von anpaßbaren Prozeduren, die eine komfortable Programmierung ermöglichen. Die Anpassung erfolgt in einem Programmpaket CONSTRL.ASM, das nach den Unterlagen des PASCAL/M an eine vorhandene Konsole adaptiert werden muß. Die Programmierung ist dabei in Assembler. Danach stehen folgende Prozeduren zur Verfügung:

PROCEDURE CONACT
(zahl: INTEGER);

Je nach dem Wert von „zahl" ist eine der folgenden acht Funktionen möglich:

CONACT(0)
Bildschirm löschen
und Cursor in die Home Position
CONACT(1)
Löschen der aktuellen Zeile
beginnend bei Cursor
CONACT(2)
Cursor eine Zeile nach oben
CONACT(3)
Cursor eine Zeile nach unten
CONACT(4)
Cursor ein Zeichen nach links
CONACT(5)
Cursor ein Zeichen nach rechts
CONACT(6)
Löschen des Zeichens
an der Cursor-Stelle
CONACT(7)
Einfügen eines Leerzeichens
an der Cursor-Stelle

PROCEDURE GOTOXY
(xkoord,ykoord : INTEGER);

Der Cursor wird wie beim UCSD-PASCAL an xkoor,ykoor gesetzt.

PROCEDURE READXY
(VAR xkoord,ykoord : INTEGER);

Die aktuelle Position des Cursors läßt sich damit ermitteln. 0,0 ist dabei in der linken Ecke oben.

PROCEDURE SCREEN
(VAR datei:FILE;
VAR zeile, spalte :INTEGER);

Die Anzahl der verfügbaren Zeilen und Spalten wird in „zeile" und „spalte" gestellt. Die Werte von CONSOLE:, CRT: und CON: sind in CONSTRL.ASM ein-

gestellt. Für andere Dateien gilt die Printer Definition in CONSTRL.ASM.

2.2.6 Weitere Fähigkeiten

In der CASE-Anweisung kann mit dem neuen Begriff OTHERWISE der Fall abgefangen werden, wenn keiner der einzelnen Fälle zutrifft.

Beispiel:

```
CASE × OF
   1: WRITE(1);
   2: WRITE(2);
   3: WRITE(3)
OTHERWISE
   WRITE('nicht 1 bis 3')
END
```

Segment-Prozeduren können wie im UCSD-PASCAL aufgebaut werden.

Um Maschinenprogramme mit verwenden zu können, gibt es die Möglichkeit mit EXTERNAL zu arbeiten. EXTERNAL wird wie FORWARD angegeben. Außerdem ist es möglich, die Maschinenunterprogramme zu numerieren.

Beispiel:

```
PROCEDURE
maschine (× : INTEGER);
   EXTERNAL 1;
```

Das Maschinenunterprogramm wird in Assembler geschrieben und muß mit einem festgelegten Kopf beginnen.

```
DB    Anzahl der vorhandenen
      Prozeduren
DW    Start des Maschinen-
      programms
JMP   Sprung auf die Prozedur 1
JMP   Sprung auf die Prozedur 2
...
Start der einzelnen Programme
```

Parameter werden im Maschinenstack übergeben. Dabei liegt der erste Parameter des Prozedurkopfes im tiefen Teil des Stacks. Der letzte Parameter kann mit einem POP-Befehl zuerst geholt werden.

Beispiel:

```
PROCEDURE
maschine( a,b,c : INTEGER;
VAR d : REAL);
```

der Stack sieht dann wie folgt aus:

```
Adresse von d
Wert c
Wert b
Wert a
```

Als erstes kann mit einem POP-Befehl die Adresse des Wertes d geholt werden. Nur eine Adresse, da es sich um einen VAR-Parameter handelt. Die Darstellung der einzelnen Variablen ist wie folgt:

Integer:

```
Byte 1        Byte 2
im Zweierkomplement
--- Rest des Stacks ---
```

Real-Format:

```
m1            exp
m2            m3
--- Rest des Stacks ---
```

Dabei sind m1 bis m3 die Mantisse und exp der Exponent. Die Darstellung des Exponents ist je nach Version folgendermaßen:

```
exp = Meeeeeee Standard
   mit M=Mantissenvorzeichen
         e=Exponentenstelle (+bias)
exp = MEeeeeee bei AM 9511 Version
         E=Exponentenvorzeichen
```

Long_Integer:

```
Byte3         Byte4
Byte1         Byte2
--- Rest des Stacks ---
```

Pointer:

```
Byte1         Byte2
--- Rest des Stacks ---
```

Char:

```
00000000 Zeichen        8 Bit-Daten
--- Rest des Stacks ---
```

Das Maschinenprogramm wird beim Start des PASCAL-Programms durch Angabe eines Parameters mitgeladen.

Das Programm wird als Datei.EXT abgelegt. Beispiel:

A)PRUN pascalprogramm
        @m=maschinenprogramm

PRUN ist dabei der Interpreter.

Weitere Aspekte sind in dem Manual [10] nachzulesen.

2.3 PASCAL/Z

PASCAL/Z ist ein Compiler, der auf dem CP/M-Betriebssystem läuft [3,11] und einen Z80 als Prozessor benötigt. Dieser Compiler erzeugt keinen P-Code, sondern direkt Z80-Maschinenbefehle. Der Compiler besitzt einige interessante Vorteile. Der Code ist ROMABLE, das heißt, er kann in einem Festwertspeicher abgelegt werden und er ist reentrant, dies ist wichtig für Multitasking oder einfachen Interrupt-Betrieb. Der erzeugte Code kann sehr klein sein (einige 100 Bytes). Es wird ein Z80-Assemblercode erzeugt, so daß Optimierungen von Hand auch möglich sind. PASCAL/Z besitzt auch Erweiterungen wie bei den UCSD-Varianten. Es gibt einen String-Typ, Direkt-Zugriff auf Dateien, damit ist es möglich, skalare Typen (siehe 1.4.1) auch in der WRITE-Anweisung direkt auszugeben; es erscheint der symbolische Name als Ausgabe. Funktionen können auch strukturierte Typen als Ergebnis zurückliefern. PASCAL/Z besitzt auch ein paar Einschränkungen, so ist z. B. GET und PUT nicht implementiert. Prozeduren oder Funktionen können nicht als Parameter verwendet werden (dies ist auch im UCSD-PASCAL nicht möglich). Die Speicherverwaltung wird mit den Funktionen NEW, MARK und RELEASE ausgeführt und nicht mit NEW und DISPOSE (ebenfalls wie im UCSD-PASCAL). Das System sollte 64K Bytes besitzen, um vernünftig mit dem PASCAL arbeiten zu können (48K bis 56K mindestens).

2.3.1 Skalare Daten-Typen

Eine interessante Erweiterung ist die folgende:

```
TYPE
   farbe = ( rot, gruen, blau );
VAR
   farbart : farbe;
```

Es wird der neue Typ „farbe" definiert; „farbart" ist eine Variable, die die Werte rot, grün und blau annehmen kann. In PASCAL/Z ist nun folgende Anweisung möglich:

```
      WRITE(farbart);
oder  WRITELN(farbart);
```

Besitzt „farbart" den Wert „gruen", so erscheint der Text GRUEN auf dem Bildschirm. Dies ist im Standard-PASCAL nicht möglich.

2.3.2 String-Verarbeitung

Ein String wird mit dem reservierten Wort STRING definiert.

Beispiel:

```
VAR
   name : STRING 80;
```

Es wird die Variable „name" definiert, die ein String mit maximal 80 Zeichen sein kann. Die maximale Angabe kann 255 sein. Eine eingebaute Prozedur ermöglicht es, einen String an einen anderen anzufügen.

```
APPEND(name, ' MUELLER');
```

Der String ' MUELLER' wird an den String „name" angehängt. Es gibt noch ein paar andere String-Funktionen, die in einer Bibliothek vorhanden sind, aber zuvor im PASCAL-Programm definiert werden müssen.

```
TYPE
   $STRING0 = STRING 0;
   $STRING255 = STRING 255;
```

```
FUNCTION LENGTH
  (x : $STRING255) : INTEGER;
  EXTERNAL;
FUNCTION INDEX
  (x,y : $STRING255) : INTEGER;
  EXTERNAL;
PROCEDURE SETLENGTH
  ( VAR x : $STRING0;
  y : INTEGER); EXTERNAL;
```

Mit LENGTH läßt sich die aktuelle Länge eines Strings ermitteln, INDEX liefert die Stelle in einem String x, bei der der Teilstring y gefunden wurde, ansonsten 0. Mit SETLENGTH läßt sich die aktuelle Länge eines Strings x auf y setzen.

Bei Stringvergleichen darf auf der linken Seite keine String-Konstante erscheinen. Es ist möglich mit einem Index, der in eckigen Klammern angegeben wird, ein einzelnes Zeichen in einem String anzusprechen.

```
name[n]
```

Es wird das n-te Zeichen angesprochen. n muß dabei zwischen 1 und der maximalen Länge von „name" liegen.

2.3.3 Datei-Verarbeitung

Die Funktionen RESET und REWRITE sind verfügbar und im Prinzip genauso definiert wie im UCSD-PASCAL, nur daß bei RESET der Dateiname nicht weggelassen werden darf und die Reihenfolge von Dateinamen und Dateivariable vertauscht ist. Beispiel:

```
RESET('A:TEXT.DAT',datei);
REWRITE('LST:',datei1);
```

Die Definition des Dateinamens entspricht der von CP/M. Als Gerät kann auch CON: und LST: verwendet werden.

GET und PUT existieren nicht, wohl aber READ, WRITE, READLN und WRITELN, gemäß dem Standard. In der Programmüberschrift darf INPUT und OUTPUT nicht angegeben werden, also

```
PROGRAM name;
```

ist eine gültige Überschrift. Es dürfen auch Dateivariablen nicht in der Überschrift erscheinen. Das Syntaxdiagramm sieht daher wie folgt aus:

```
PROGRAM programmname ;
  block.
```

In PASCAL/Z ist es auch möglich, einen Direkt-Zugriff durchzuführen. Dazu wurde die Syntax für READ und WRITE erweitert.

```
    WRITE( dateivariable :
    rekordzahl, rekordvariable);
und READ( dateivariable :
    rekordzahl, rekordvariable);
```

Beispiel: Es soll an den 30ten Rekord der Datei „datei" der Inhalt des Rekords r geschrieben werden.

```
WRITE(datei:30,r);
```

1 ist dabei der erste Rekord einer Datei. Der Random-Access ist nur mit einer CP/M-Version 2.0 aufwärts möglich.

2.3.4 Diverse Befehle

Die CASE-Anweisung kann wie bei PASCAL/M durch OTHERWISE, hier durch einen ELSE-Teil ergänzt werden. Also z. B.:

```
CASE farbvariable OF
  rot: rotauswertung;
  gruen: gruenauswertung;
ELSE: restauswertung
END;
```

Einführen von Maschinenunterprogrammen:

Dies geschieht mit der EXTERNAL-Anweisung, die schon vorher einmal verwendet wurde. Beispiele:

```
FUNCTION mix
  (par1,par2 : farbe): farbe;
  EXTERNAL;
```

```
B>type testpas.pas
program test;
   var
      i:integer;

   function fac(i:integer):integer;
      begin
         if i = 0 then fac := 1
            else fac := fac(i-1)*i
      end;


begin
   writeln('Fakultaet ');
   for i:=1 to 6 do
      writeln('fac von ',i,' ist ',fac(i))
end.
```

Abb. 2.3.4-1
Beispielprogramm
für PASCAL/Z

```
pascal testpas.bbb
InterSystems Pascal ver 3.2-1
TEST            1-
FAC             5--
TEST           12-
```

Abb. 2.3.4-2
Übersetzerstart

```
B>type testpas.lst

TEST                                        Page 1

    1  program test;
    1     var
    1        i:integer;
    1
    1     function fac(i:integer):integer;
    1        begin
    1           if i = 0 then fac := 1
    3              else fac := fac(i-1)*i
    4        end;
    4
    4
    4  begin
    4     writeln('Fakultaet ');
    5     for i:=1 to 6 do
    6        writeln('fac von ',i,' ist ',fac(i))
    7  end.
```

Abb. 2.3.4-3
Ausgabeprotokoll

```
type testpas.src
L138
         ENTR    D,2,0
         STMT    D,1
         MOV     L,8(IX)
         MOV     H,9(IX)
         MOV     D,A
         MOV     E,A
         DSB1    D,0
         JNZ     L140
         STMT    D,2
         MOV     3(IX),A
         MVI     2(IX),1
         JMP     L153
L140
         STMT    D,3
         MOV     L,8(IX)
         MOV     H,9(IX)
         DCX     H
         PUSH    H
         CALL    L138
         STMT    M,3
         MOV     L,8(IX)
         MOV     H,9(IX)
         MULT    D,0
         MOV     3(IX),H
         MOV     2(IX),L
L153
         EXIT    D,2
L99
         ENTR    D,1,2
         STMT    D,4
         JR      L174
L173
         DB      ' teatlukaF',10
L174
         LXI     H,769
         PUSH    H
         LXI     B,10
         PUSH    B
         LXI     H,-10
         DADD    S
         SPHL
         XCHG
```

Abb. 2.3.4-4
Erzeugter Assembler-Code

```
            LXI     H,L173+0
            LXI     B,10
            LDIR
            LXI     B,14
            CALL    L109
            STMT    M,4
            STMT    D,5
            MOV     0(IY),A
            MVI     -1(IY),1
            PUSH    IY
            LXI     H,6
            XTHL
L183
            MOV     D,M
            DCX     H
            MOV     E,M
            XTHL
            PUSH    H
            GE      D,0
            JNC     L184
            STMT    D,6
            JR      L198
L197
            DB      ' nov caf',8
L198
            LXI     H,769
            PUSH    H
            LXI     B,8
            PUSH    B
            LXI     H,-8
            DADD    S
            SPHL
            XCHG
            LXI     H,L197+0
            LXI     B,8
            LDIR
            LXI     H,522
            PUSH    H
            MOV     L,-1(IY)
            MOV     H,0(IY)
            PUSH    H
            JR      L212
L211
            DB      ' tsi ',5
```

zu Abb. 2.3.4-4

```
L212
        LXI     H,769
        PUSH    H
        LXI     B,5
        PUSH    B
        LXI     H,-5
        DADD    S
        SPHL
        XCHG
        LXI     H,L211+0
        LXI     B,5
        LDIR
        MOV     L,-1(IY)
        MOV     H,0(IY)
        PUSH    H
        CALL    L138
        STMT    M,6
        PUSH    D
        LXI     H,522
        XTHL
        PUSH    H
        LXI     B,29
        CALL    L109
        STMT    M,6
        CTRL
        POP     H
        XTHL
        INR     M
        INX     H
        JRNZ    L225
        INR     M
        JV      L226
L225
        JMP     L183
L184
        POP     D
L226
        POP     D
        FINI
```

zu Abb. 2.3.4-4

```
B>asmbl main,testpas.bb/rel
Pascal/Z run-time support interface                  ASMBLE v-5j

0 errors.  223 symbols generated.  Space for 4327 more symbols.
4027 characters are stored in 43 macros.
579 bytes of program code.
link /n:b:testpas b:testpas /e
LINK version 1m
Load mode
Generate a COM file
Lo = 0100    Hi = 0ED5    Start = 0108    Save  14 blocks
```

Abb. 2.3.4-5 Assemblierung und Binden

```
testpas
Fakultaet
fac von          1 ist          1
fac von          2 ist          2
fac von          3 ist          6
fac von          4 ist         24
fac von          5 ist        120
fac von          6 ist        720
```

Abb. 2.3.4-6 Starten des eigentlichen Programms

```
TESTPAS BAK   1K
TESTPAS COM   4K
TESTPAS LST   1K
TESTPAS PAS   1K
TESTPAS REL   2K

TESTPAS SRC   2K
```

Abb. 2.3.4-7 Diskettenprotokoll der Dateien

Das dazugehörige Assemblerprogramm sieht dann wie folgt aus:

```
     ENTRY mix ;name für den Linker
mix: pop h     ;rücksprungadresse
     pop d     ;holen von par1,par2
     mov a,e
     add d
     mov e,a
     xra a
     mov d,a   ;ergebnis in de
     pchl      ;rücksprung
```

Hierzu gibt es noch viel zu sagen, was aber den Umfang des Buches sprengen würde, siehe dazu [3,4,11].

Eine Segment-Prozedur gibt es nicht, aber einen ähnlichen Mechanismus um Programm nachladen zu können. Mit der nachfolgenden Anweisung kann eine Verkettung (chain) von Programmen durchgeführt werden:

FTXTIN(programmname); CHAIN;

Der Programmname kann ein String in Anführungszeichen oder ein Feld vom Typ ARRAY [1..n] OF CHAR sein. Beim Linker muß dann der CHAIN-Modul, der in der Bibliothek steht, dazugebunden werden. Es soll hier einmal gezeigt werden, wie ein kompletter Übersetzungsvorgang durchgeführt wird. *Abb. 2.3.4-1* zeigt das Quellprogramm und *Abb. 2.3.4-2* die Ausgabe bei dem Übersetzungsvorgang, die auf der Konsole er-

scheint. Es wird dabei ein Listing auf Diskette erzeugt, wie es *Abb. 2.3.4-3* darstellt. Gleichzeitig wird die Assemblersource erzeugt. Diese ist in *Abb. 2.3.4-4* abgebildet. Es sind einige Makros verwendet, die in einer Bibliothek vorhanden sind. Es wird dann der Assembler gestartet und anschließend der Binder (Linker), der noch einige Unterprogramme aus einer Bibliothek dazubindet. *Abb. 2.3.4-5* zeigt den Ablauf; in *Abb. 2.3.4-6* ist schließlich der Gesamtablauf des Programms abgebildet. Das Programm wird durch Angabe seines Namens (testpas) gestartet und ist als COM-Datei abgelegt [3]. Wie ein Übersetzer arbeitet ist in [5] beschrieben. *Abb. 2.3.4-7* zeigt die Dateien.

2.4 PASCAL/MT

PASCAL/MT [12] ist ebenfalls ein Compiler, der Maschinencode erzeugt. Er ist speziell für System-Programme geeignet. Der Compiler läuft unter CP/M[3] in minimal 32K Bytes (Version 3.0). Es ist ein symbolischer Debugger vorhanden, mit dem es möglich ist, die Programme komfortablel zu testen. Das erzeugte Programm benötigt einen minimalen Platz von 1.25K Bytes. CP/M-Dateizugriff ist auf der Block-Ebene möglich. Assembler-Code kann direkt in die PASCAL-Programme geschrieben werden. Es kann zwischen einem Standard-Gleitkommapaket oder einem BCD-Festkommapaket gewählt werden. Einige Nachteile sind, daß Mengen nicht verfügbar sind und GET und PUT nicht vorhanden ist. Will man rekursive Funktionen oder Prozeduren anwenden, muß dies durch einen besonderen Befehl erst ermöglicht werden.

2.4.1 Maschinenprogramme

Mit einer EXTERNAL-Anweisung können externe Maschinenunterprogramme aufgerufen werden. Es können maximal drei Parameter an das Unterprogramm übergeben werden. Dies geschieht in den Registern BC DE und HL. Wird ein Parameter mit VAR angegeben, so wird die Adresse übergeben.
Beispiel:

```
PROCEDURE EXTERNAL[5] BDOS
(funktion : INTEGER;
VAR adresse : INTEGER);
PROCEDURE EXTERNAL[$FC00]
motor(n,pos : INTEGER);
```

Eine andere Möglichkeit ist es, direkt Assemblerprogramm in den PASCAL-Quell-Text einzufügen. Dazu steht die INLINE-Prozedur zur Verfügung. Marken sind auch möglich. Beispiel:

```
PROCEDURE demo_inline;
  VAR
    i : INTEGER;
  BEGIN
    INLINE( [marke] "lxi h /$fc00/
                    "shld /i/
                    "lda /$f000/
                    "ora a
                    "jnz /marke/ );
  END;
```

Mit dem Befehl INLINE lassen sich auch Felder vorbelegen. Dazu gibt es noch eine Funktion ADDR( ) mit der sich die Adresse von Marken oder anderen Objekten bestimmen läßt. Die 8080-Befehle sind in [3,4] ausführlich beschrieben.

2.4.2 Datei-Verarbeitung

Die Prozeduren READ, READLN, WRITE und WRITELN sind für die Konsole definiert, nicht aber für Dateien. Nur bei WRITE und WRITELN kann eine Ausgabe auf den Drucker verlangt werden:

```
WRITE(PRINTER,
' Ausgabe auf Drucker ');
```

Für den Diskettenzugriff gibt es spezielle Prozeduren:

```
OPEN(fcbname,titel,ergebnis);
OPEN(fcbname,titel,ergebnis,
  extent_zahl);
CLOSE(fcbname,titel,ergebnis);
DELETE(fcbname);
BLOCKREAD
  (fcbname,buffer,ergebnis);
BLOCKREAD
  (fcbname,buffer,ergebnis,
  relativblock);
BLOCKWRITE
  (fcbname,buffer,ergebnis);
BLOCKWRITE
  (fcbname,buffer,ergebnis,
  relativblock);
```

Dabei ist

fcbname: eine Variable vom typ TEXT oder FILE

titel: ARRAY [0..1] OF CHAR;
titel[0] = disk auswahlbyte;
(· 0= aktuelle Login-Drive
1= Drive A: ,2= Drive B: ... ·)
titel[1] .. [8] = dateiname mit 8 Zeichen
titel[9] .. [11] = extension mit 3 Zeichen

ergebnis: Wert der vom BDOS zurückgeliefert wird (siehe CP/M [3])

buffer: Start eines 128 Byte Feldes (norm. ARRAY ... OF CHAR)

relativblock:
Integerzahl 0..255 für Direktzugriff.

Eine Variable vom Typ TEXT oder FILE ist hier ein feld 0..35 OF CHAR und entspricht den BDOS-Definitionen für Datei-Verwaltung.

2.4.3 Diverse Befehle

Es gibt in PASCAL/MT einige Operationen, die eine schnelle Ausführung von speziellen Aufgaben ermöglichen.

```
PROCEDURE MOVE
(quelle,ziel : feldname oder
Integer; länge : INTEGER);
```

Der Feldname oder Integer wird als Zeiger für das Ziel oder die Quelle verwendet.

```
PROCEDURE EXIT
```

Die aktuelle Prozedur oder Funktion wird verlassen.

```
PROCEDURE TSTBIT
(zahl : 16bitvariable;
bitnr : INTEGER):BOOLEAN;
```

Als 16bitvariable kann ein Typ INTEGER, CHAR oder BOOLEAN treten. Die „bitnr“ kann von 0 bis 15 reichen.

```
PROCEDURE SETBIT
(VAR zahl : 16bitvariable;
bitnr : INTEGER);
```

Das entsprechende Bit in „zahl“ wird gesetzt.

```
PROCEDURE CLRBIT
(VAR zahl : 16bitvariable;
bitnr : INTEGER);
```

Das Bit an der Stelle „bitnr“ wird zurückgesetzt.

```
FUNCTION SHR
(zahl : 16bitvariable;
bitnr: INTEGER) : 16bitvariable;
FUNCTION SHL
(zahl : 16bitvariable;
bitnr : INTEGER) : 16bitvariable;
```

Hiermit ist ein Links- oder Rechtsschieben möglich.

```
FUNCTION LO
(zahl : 16bitvariable) : 16bitvariable;
```

Die niederwertigen 8 Bits werden als Ergebnis übermittelt.

```
FUNCTION HI
(zahl : 16bitvariable) : 16bitvariable;
```

Die höherwertigen 8 Bits werden in die unteren 8 Bits gestellt.

```
FUNCTION SWAP
(zahl : 16bitvariable) : 16bitvariable;
```

Die unteren 8 Bits werden mit den höherwertigen 8 Bits vertauscht.

```
FUNCTION ADDR
(name : variablenreferenz)
: 16bitvariable;
```

Die Maschinenadresse der Variablen läßt sich damit ermitteln. Es können auch Prozedur-, Funktions- indizierte-Variablen oder Rekordnamen verwendet werden.

```
PROCEDURE WAIT
(portnummer : constante;
maske : constante; polaritaet :
BOOLEAN);
```

Dadurch wird folgender Kode erzeugt:

```
IN portnummer
ANI maske
J?? $-4
```

?? ist Z wenn die Polarität FALSE ist
?? ist NZ wenn die Polarität TRUE ist.

```
FUNCTION SIZEOF
(name :variablen oder
typname) : INTEGER;
```

Die Anzahl der von „name" belegten Variablen wird ermittelt. Damit läßt sich die NEW-Funktion nachbilden, die standardmäßig nicht vorhanden ist.

```
VAR heappointer : INTEGER;
   p : ↑symboltabelleneintrag;
...
(· neuen Pointer aufbauen ·)
p := heappointer;
heappointer := heappointer +
SIZEOF(symboltabelleneintrag)
```

2.4.4 Debugger

Ein symbolischer Debugger kann beim Austesten von PASCAL/MT-Programmen verwendet werden. Wird normalerweise ein PASCAL-Programm getestet, so werden WRITELN-Anweisungen in die Quelle geschrieben, um z. B. verschiedene Variablen-Werte während des Programmlaufs auszugeben. Mit dem symbolischen Debugger ist es in PASCAL/MT möglich, sich die Werte auch ohne Veränderung der Source zu beschaffen. Auch kann ein Einzelschritt und Tracelauf durchgeführt werden. Bei der Übersetzung muß angegeben werden, ob der Debugger mit eingebunden werden soll oder nicht.

Go mit optionalem Breakpunkt
Die Programmausführung kann bis zu einem bestimmten Breakpunkt fortgesetzt werden. Der Breakpunkt wird durch Angabe einer Zeilennummer oder den Prozedur- oder Funktionsnamen angegeben.

Trace
Ein oder mehrere Zeilen werden ausgeführt.

Prozedur- und Funktionenanzeige
Es ist möglich, alle aufgerufenen Funktionen oder Prozeduren anzuzeigen.
Weitere Befehle ermöglichen es, zwischen einer langsamen bis schnellen Ausführung zu wählen, um die aufgerufenen Prozeduren oder Funktionen, deren Namen ja auf dem Bildschirm erscheinen, optimal verfolgen zu können.

Ein permanenter Breakpoint kann auch gesetzt werden, so daß es möglich ist, den Programmablauf immer an der gleichen Stelle zu stoppen.

Eine der wichtigsten Möglichkeiten ist es, Variablenwerte ausgeben zu können. Beispiele:

D I	Ausgeben des Wertes der globalen Variablen I
D A:I	Ausgabe des Variablenwertes I, der in der Funktion oder Prozedur A bekannt ist.
D F1	Wert der Funktion F1 ausgeben

Bei Feldern wird automatisch nach dem Index gefragt.
Alles in allem ist auch PASCAL/MT eine interessante Version für Mikrorechner und es ist schon eine neue Release angekündigt, bei der weitere Standard-Befehle von PASCAL implementiert sind.

3 Anhang

3.1 Syntaxdiagramme

PASCAL läßt sich durch Syntaxdiagramme sehr anschaulich definieren. In *Abb. 3.1.1* sind alle Syntaxdiagramme der Sprache abgedruckt. Die Beschreibung des Gebrauchs ist in Abschnitt 1.2.1 zu lesen.

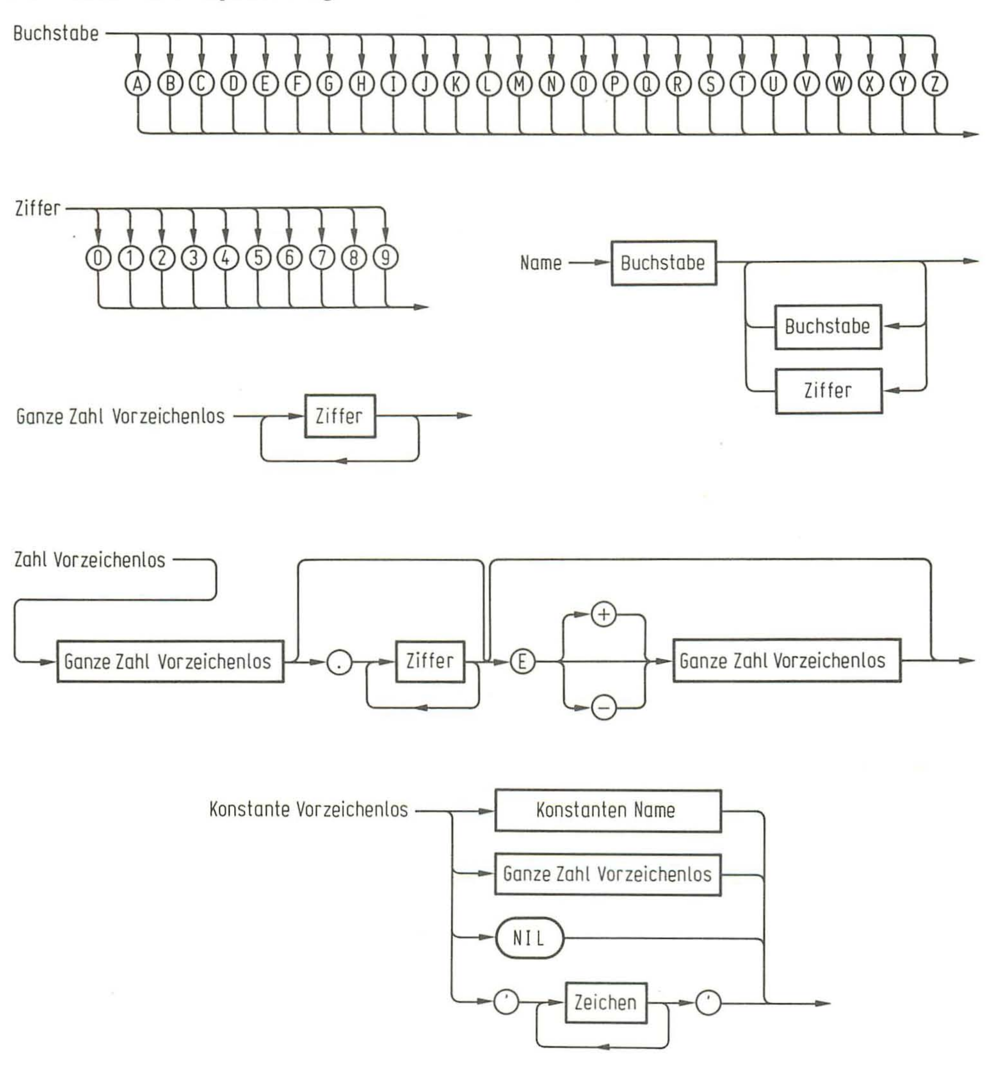


Abb. 3.1-1 Syntaxdiagramme zu PASCAL

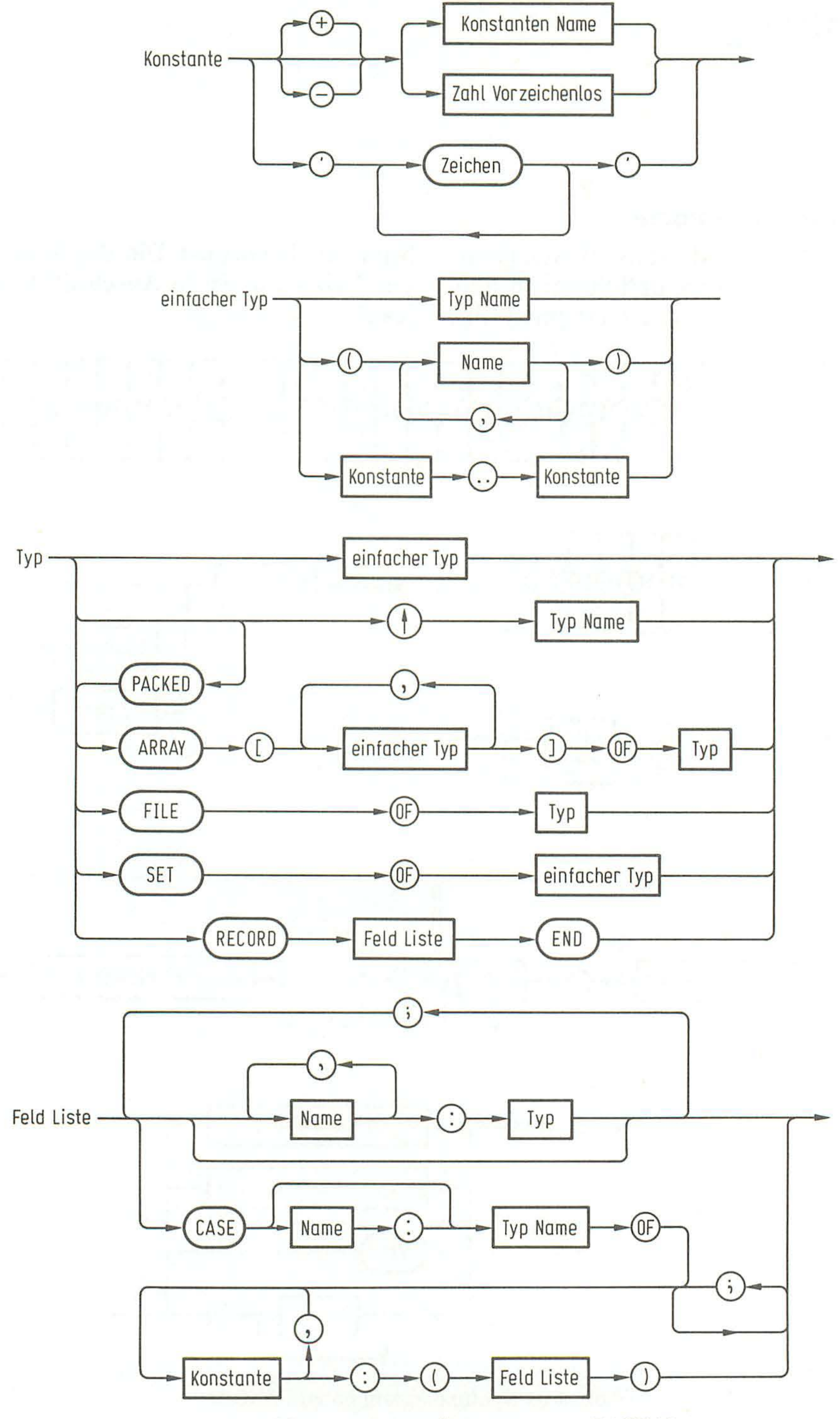


Abb. 3.1-1 Syntaxdiagramme zu PASCAL

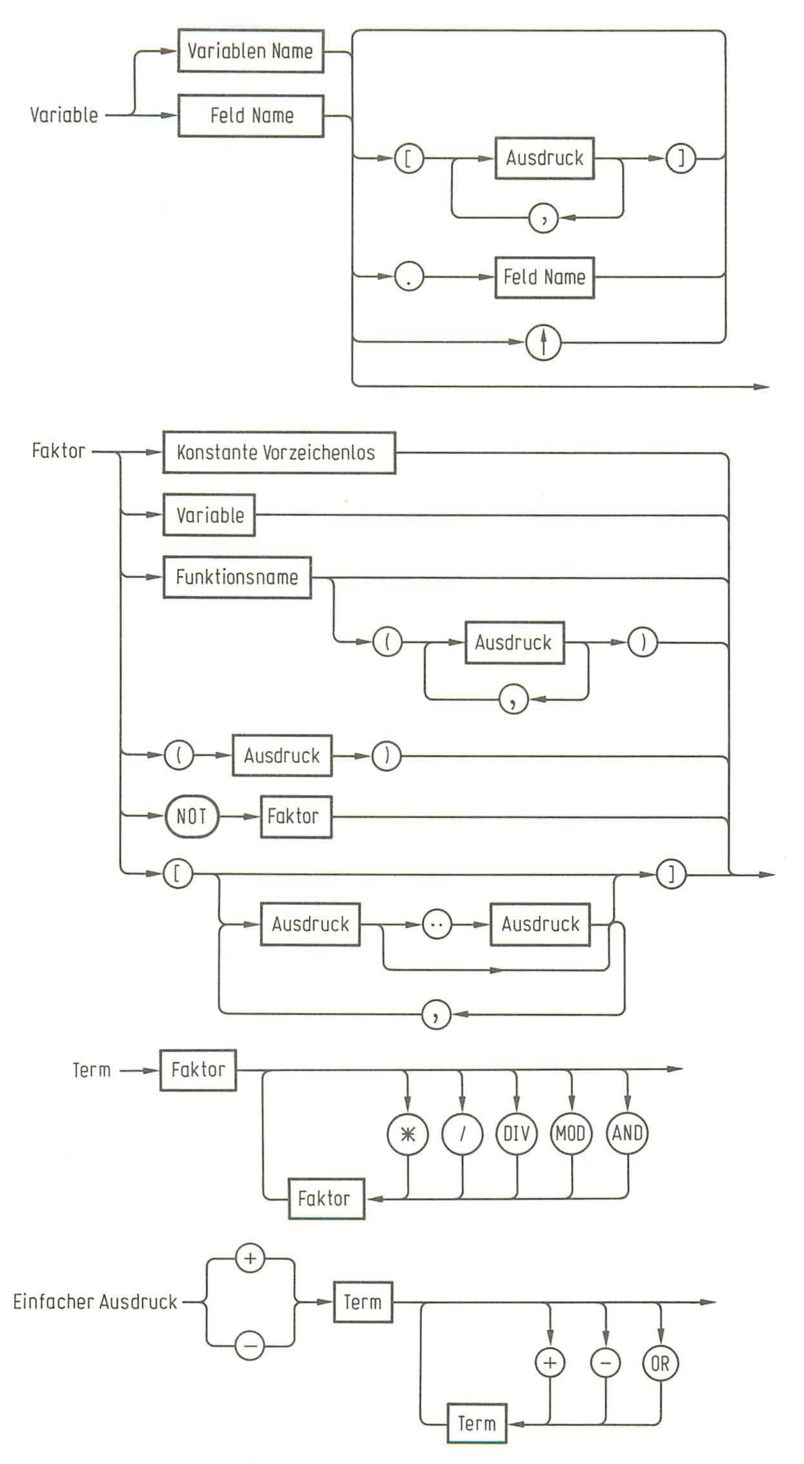


Abb. 3.1-1 Syntaxdiagramme zu PASCAL

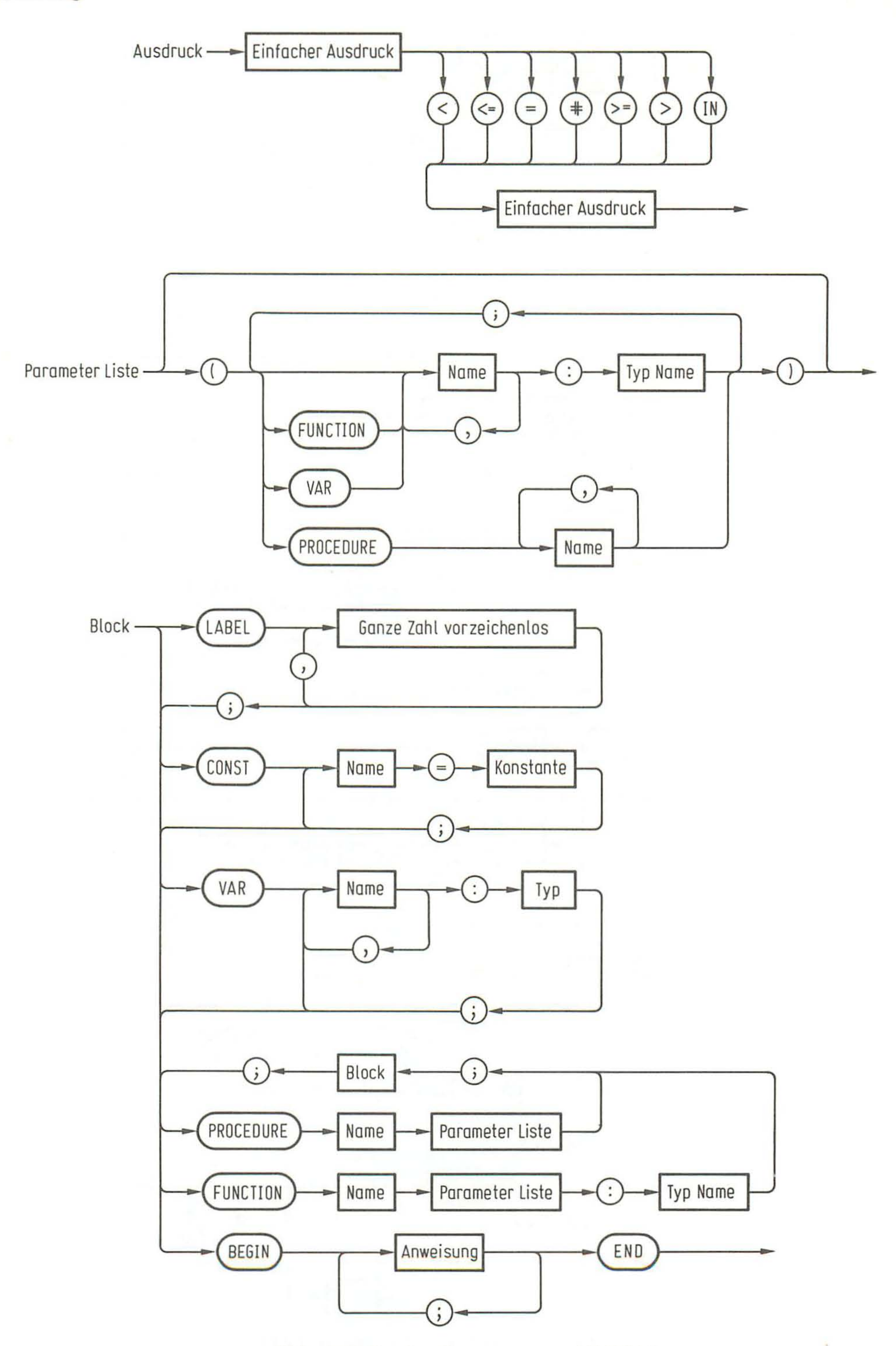


Abb. 3.1-1 Syntaxdiagramme zu PASCAL

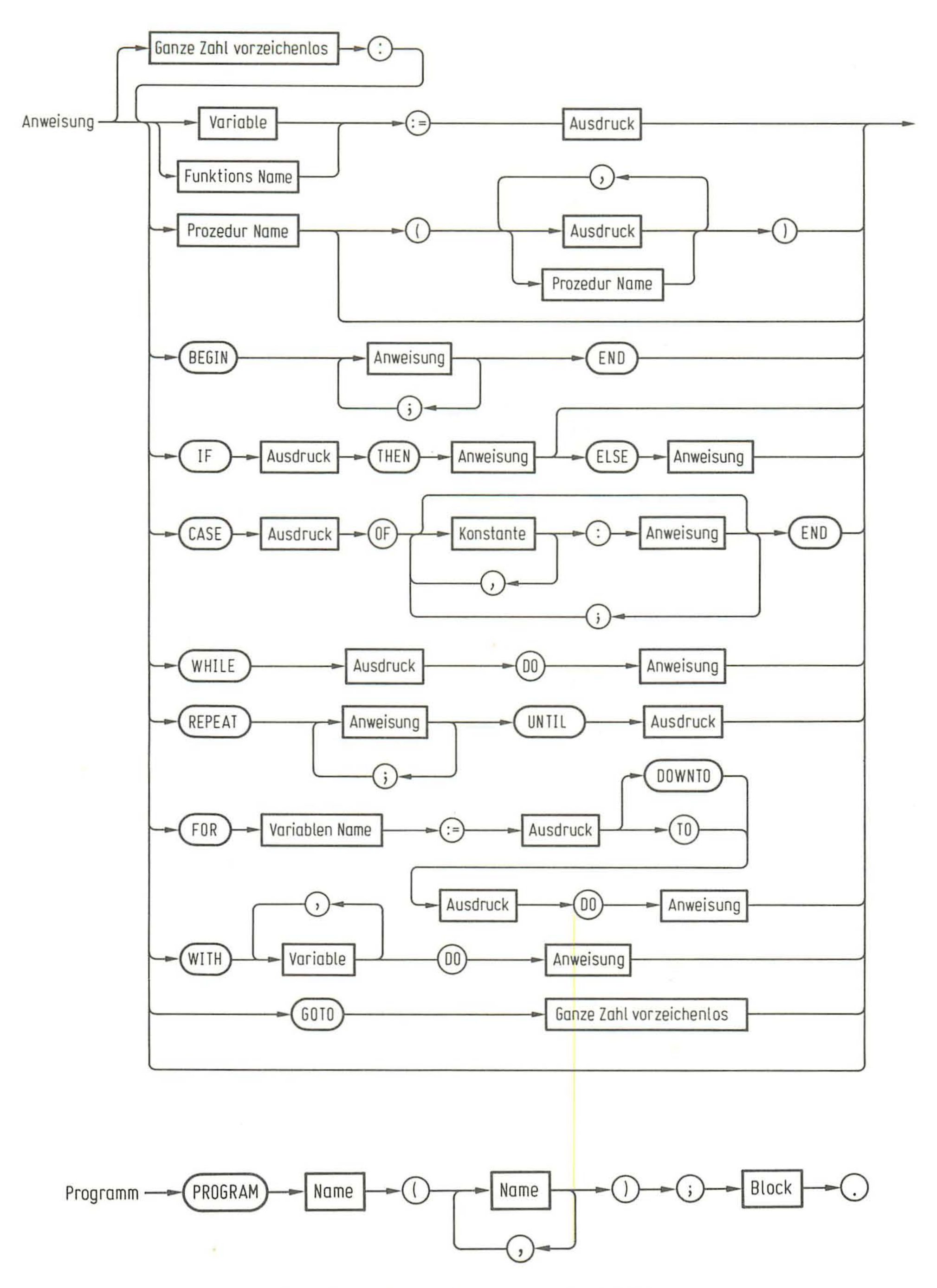


Abb. 3.1-1 Syntaxdiagramme zu PASCAL

Literatur

[1] Peter Grogono. Programming in PASCAL. ADDISON-WESLEY PUBLISHING COMPANY, INC. Menlo Park, California. ISBN 0-201-02473-X. 1978.

[2] Nikolaus Wirth, Kathleen Jensen. PASCAL USER MANUAL AND REPORT. Springer-Verlag. New York, Heidelberg, Berlin. ISBN 0-387-90144-2. 1975.

[3] Rolf-Dieter Klein. Mikrocomputer Hard- und Softwarepraxis. FRANZIS-VERLAG, 1981. ISBN 3-7723-6811-5.

[4] Rolf-Dieter Klein. Mikrocomputersysteme. FRANZIS-VERLAG, 1979. ISBN 3-7723-6382-2.

[5] Rolf-Dieter Klein. BASIC-Interpreter. FRANZIS-VERLAG. ISBN 3-7723-6941-3.

[6] Plate. PASCAL. FRANZIS-VERLAG. ISBN 3-7723-6901-4.

[7] Herwig Feichtinger. Basic für Mikrocomputer. FRANZIS-VERLAG, 1980. ISBN 3-7723-6821-2.

[8] APPLE-PASCAL REFERNENCE MANUAL. APPLE COMPUTER INC. 1979.

[9] North Star Pascal System Reference Manual. North Star Computers, Inc. 1979.

[10] PASCAL/M USER'S REFERENCE MAUAL. Digital Marketing. Calif. 1979.

[11] PASCAL/Z IMPLEMENTATION MANUAL 3.2. Ithaca Intersystems, Inc. 1980.

[12] PASCAL/MT Release 3.0 User's Guide. MicroSYSTEMS. 1980.

[13] Peter Wegner. Programming with ADA: An Introduction by means of graduated examples. Departement of Computer Science Brown University.

Terminologie

A

ABS
Der Absolutbetrag einer Zahl,
z. B. ABS(−5)=ABS(5)

AND
Die logische Und-Funktion. Dabei können nur boolesche Argumente verwendet werden

ARCTAN
Die Unkehrfunktion von TAN.

ARRAY
Feld

B

BASIC
Beginners All Purpose Symbolic Instruction Code, eine weit verbreitete niedere Programmiersprache

BEGIN
Start eines Blocks

BOOLEAN
eine Variable die nur zwei Werte annehmen kann, 0 oder 1 bzw. TRUE oder FALSE

C

CASE
Möglichkeit um Fallunterscheidungen zu verwirklichen

CHAIN
Nachladen von Programmen
nicht im Standard-Pascal

CHAR
Damit lassen sich Zeichen darstellen

CHR
Umkehrfunktion zu ORD(×)

Compiler
Übersetzer, der eine Sprache in einen Maschinencode übersetzt

CONST
Deklaration von Konstanten folgt

COS
Cosinusfunktion. Der Parameter wird als in Radiant gegeben interpretiert

Cross-Assembler
Ein Übersetzungsprogramm, das nicht auf dem Rechner läuft für das es Code erzeugt

D

Debugger
Hilfsmittel zur Fehlersuche

DISPOSE
Freigabe von Speicherzellen

DIV
Ganzzahlige Division in Pascal. Ein entstandener Rest geht verloren

DO
Teil der WHILE-Anweisung
Teil der FOR-Anweisung
Blockanfang

DOWNTO
Teil der FOR-Anweisung für Rückwärts laufende Indizes

E

EDITOR
Ein Programm zur Eingabe von Texten (Programmen)

ELSE
Alternative bei der IF-Anweisung

END
Ende eines Blocks

EOF
Feststellen ob Dateiende vorliegt

EOLN
Prüft, ob das Zeilenende erreicht wurde

EXP
Exponentialfunktion

F

FILE
Schlüsselwort für Datei

Floppy
flexibles magnetisches Speichermedium für kleine Systeme. Kapazität von 79 Kbyte bis 5 Mbyte

FOR
Schleifenkonstruktion mit Indexvariable

FORTRAN
Eine Programmiersprache, die besonders für wissenschaftliche Berechnungen verwendet wird

FORWARD
Möglichkeit eine Prozedur anzukündigen ohne den detailierten Ablauf anzugeben

FUNCTION
Definition einer Funktion in Pascal

G

GET
Lesen von einer Datei

GOTO
Sprunganweisung

H

Hardware-Compiler
Eine Sprache die es ermöglicht, als 'Code' eine Hardware (Schaltungen) zu erzeugen

I

IF
Bedingte Anweisung

IN
Eine Operation für Mengen, die entspricht dem bekannten ELEMENT VON

INPUT
Standard Eingabe

INTEGER
Ganze Zahlen im Bereich – n bis + n, wobei n die maximale Zahl ist, die darstellbar ist

Interpreter
Er führt einen Code (Sprache), der auf dem Rechner nicht vorhanden ist direkt aus (langsam)

ISO-Pascal
PASCAL-Standard, der einige Erweiterungen gegenüber der Version von Prof. Wirth enthält

K

Kommentar
kann im Programm nach belieben eingefügt werden und soll häufig angewandt werden

Konstante
sind Werte, die sich beim Ablauf nicht ändern

L

Lader
Ein Programm, das ein vom Übersetzer erzeugtes Objektprogramm für den Rechner startbereit macht

LN
natürlicher Logarithmus

LOKALE Variable
Variable, die nur in einem Bereich gültig sind, außerhalb des Bereiches nicht mehr

M

MARK
Festhalten einer Position für nachfolgendes RELEASE

MOD
Modulo-Funktion in Pascal zur Ermittlung eines Restes bei der Division

N

NEW
Erzeugen eines neuen Objektes

NOT
Die logische Nicht-Funktion. NOT TRUE ergibt FALSE

O

ODD
Stellt fest, ob das Argument ungerade ist

OF
Teil der CASE-Anweisung

OR
Die logische Oder-Funktion. TRUE OR FALSE ergibt TRUE

ORD
Die ordnungszahl des Arguments wird als Ergebnis geliefert

OUTPUT
Standard Ausgabe

P

PACK
Umwandlung von einer nichtgepackten Darstellung in eine gepackte Darstellung

PACKED Aufforderung zur gepackten Darstellung

PASCAL
Die Programmiersprache PASCAL wurde von Prof. N. Wirth erfunden

PASCAL/M
Pascal Dialekt von Digital Marketing

PASCAL/MT
Pascal-Dialekt von MicroSYSTEMS, die jetzt auch einen ISO-Pascal-Compiler (PASCAL/MT+) liefern

PASCAL/Z
Pascal-Dialekt von Ithaca Intersystems

PL/1
Höhere Programmiersprache

PL/M
Ein Abkömmling von PL/1 zur Programmierung von Mikrorechnern (Intel 8080)

Platte
Magnetisches Speichermedium, Kapazität von 5 Mbyte bis 300 Mbyte

pointer
Zeiger

PRED
Der Vorgänger bezüglich der Ordnungszahl

PROCEDURE
Unterprogramm

PROGRAM
Überschrift eines jeden PASCAL-Programms

Prozedurname
Name für ein Unterprogramm, das auch Prozedur genannt wird

PUT
Schreiben auf eine Datei

R

RANDOM
Zufalls-Funktion, nicht im Standard-Pascal

READ
Einlesen

READLN
Eingabe bis zum Zeilenende

REAL
eine Gleitkommazahl

records
Datensätze

Rekursion
Der Aufruf einer Prozedur oder Funktion erfolgt direkt oder indirekt durch die Prozedur (Funktion) selbst

RELEASE
Freigabe von Speicherplatz bis zur von MARK festgelegten Position

REPEAT
Die Wiederholungsanweisung

Reservierte Wörter
Ein Name, der vom Programmierer nicht mehr verwendet wird

RESET
Zeiger auf Anfang einer Datei setzen

Resident Assembler
Ein Übersetzungsprogramm das auf dem gleichen Rechner läuft für den es auch Code erzeugt

REWRITE
Dateizeiger auf Anfang setzen

ROUND
Runden auf den nächsten ganzzahligen Wert
ROND (3.5) = 4

S

Semantik
Damit ist die Bedeutung von Sprachelementen gemeint

set
Menge

SIN
Der Sinuswert. Der Parameter wird als Radiant interpretiert

SQR
Das Quadrat (nicht Wurzel) einer Zahl SQR(3) = 9

SQRT
Die Quadratwurzel einer Zahl. SQRT(4) = 2

STRING
Definition einer Zeichenkette (nicht Standard-Pascal)

string
Zeichenkette

subrange
Unterbereich

SUCC
Der Nachfolger bezüglich der Ordnungszahl

Syntaxdiagramme
Dienen der Darstellung einer Syntax von einer Programmiersprache

Syntaxfehler
Ein Fehler, der auftritt wenn die syntaktischen Regeln nicht beachtet wurden (Syntaxdiagramm)

T

Terminalsymbol
Das Zeichen, welches der Übersetzer als Endezeichen im Zusammenhang mit der Syntax erkennt

TO
Teil der FOR-Anweisung

Trennzeichen
Symbole, die einen Namen vom anderen trennen

TRUNC
Abschneiden der Nachkommstellen. TRUNC (12.35) = 12

TYPE
Die Definition von Typen

U

UCSD-Pascal
PASCAL-Version der University California San Diego

union
Vereinigung

UNPACK
Umwandlung von einer gepackten Darstellung in eine nicht gepackte Darstellung

V

VAR
Die Deklaration von Variablen folgt

VAR-Parameter
Die Übergabe von Werten erfolgt durch Übergabe einer Adresse

Variablennamen
Der symbolische Name für eine Variable (math)

Vergleiche
Operation, mit denen die Ordnungsrelation von Werten getestet werden kann

W

WHILE
Schleifenkonstruktion mit Abbruchkriterium am Anfang

WITH
Abkürzungs-Operation für Record-Aufrufe

WRITE
Ausgabe

WRITELN
Ausgabebefehl mit nachfolgendem Wagenrücklauf und Zeilenvorschub

Z

Zeiger
Interne Darstellung, ist eine Adresse auf einen Speicherplatz

Zuweisung
Der Inhalt von einer Speicherzelle wird in eine andere übertragen

Sachverzeichnis

Weitere Franzis Elektronik-Fachbücher

Mikrocomputersysteme

Selbstbau, Programmierung, Anwendung.
Von **Rolf-Dieter Klein**
2., verbesserte Auflage. 159 Seiten mit 133 Abbildungen und 11 Tabellen. Lwstr-geb. DM 32,–
ISBN 3-7723-6382-2

Kaum zu glauben, daß ein Mikrocomputer im Selbstbau hergestellt werden kann! Daß dieses Vorhaben glückte, hat der Autor bewiesen. Wie ein hinreichend ausgebildeter Elektroniker das nachvollziehen kann, wird in dem Buch hier dargestellt.
Zunächst muß die Hardware geschaffen werden. Eingabetastatur, Mikroprozessor, Speicher verschiedener Art, Drucker, Sichtgerät, das alles muß zu einer funktionierenden Einheit zusammengeschlossen werden. Und das geht. Es geht sogar mit preiswerten, modernen Teilen, die in den einschlägigen Fachhandlungen zu haben sind.
Nun die Software. Da zeigt der Autor mehrere Möglichkeiten auf. Nicht etwa nur ein kleines Programm, das immer wieder stupide abläuft. Nein, ausführliche Programme werden vorgestellt, die zahlreiche Spiele, mathematische Aufgaben, wissenschaftliche Probleme bearbeiten können.
Als Abschluß und Höhepunkt fügt der Autor Anregungen hinzu, selbst Programme zu schreiben und in dem eigenen Mikrocomputer zu erproben. Was will man mehr?

Pascal: Einführung – Programmentwicklung – Strukturen

Ein Arbeitsbuch mit zahlreichen Programmen, Übungen und Aufgaben. Von Jürgen **Plate** und Paul **Wittstock**. 395 Seiten mit 178 Abbildungen.
Lwstr-geb. DM 48,–
ISBN 3-7723-6901-4

Das Buch könnte auch die Pascal-Fibel genannt werden. Schritt für Schritt führt es den Leser in das Programmieren mit Pascal ein. Die Autoren haben sich echt in die Ahnungslosigkeit des Anfängers hineinversetzt. Sie bringen ihm das besondere Denken des routinierten Programmierers bei. Das Verblüffende dabei ist, sie kommen mit einer einfachen klaren Sprache aus, verabscheuen das EDV-Chinesisch, setzen nichts voraus, können wunderbar erklären. Wer sich an dieses Buch heranmacht, meint, es gäbe nichts einfacheres als Pascal.

Aus dem Inhalt:
Einführung. Elemente von Pascal, Grundlagen. Einfache Kontrollstrukturen. Variable, Konstante und Arithmetik. Eingabe und Ausgabe. Programmentwicklung. Prozeduren und Funktionen (Unterprogramme). Typen. Mengen. Records. Files. Dynamische Strukturen. Text und Dokumentationshilfen. Interaktiver Verkehr. Ausflug in die Hardware. Ausblick. Anhang: Pascal Syntax. Fehlermeldung des Compilers.

Basic für Mikrocomputer

Geräte – Begriffe – Befehle – Programme.
Von **Herwig Feichtinger**
256 Seiten mit 40 Abbildungen.
Lwstr-kart. DM 26,–
ISBN 3-7723-6821-2

Dieses praxisorientierte Buch ist Einführung und Nachschlagewerk zugleich. Begriffe aus der Computer-Fachsprache wie ASCII, RS-232-Schnittstelle oder IEC-Bus werden ebenso ausführlich erläutert wie alle derzeit üblichen Befehlsworte der Programmiersprache Basic. Marktübliche Basic-Rechner werden einander gegenübergestellt – einerseits, um vor dem Kauf die Qual der Wahl zu erleichtern, andererseits um das Anpassen von Programmen an den eigenen Rechner zu ermöglichen. Und schließlich findet der Leser handfeste Tips für das Erstellen eigener Programme und Beispiele fertiger Problemlösungen für typische Anwendungsfälle.
Der Autor befaßt sich seit etwa 1975 mit der Mikrocomputer-Technik. Im Rahmen seiner Tätigkeit als FUNKSCHAU-Redakteur arbeitete er mit den meisten der hier beschriebenen Geräte selbst, was einen objektiven Vergleich möglich machte.

Franzis-Verlag München

Klein
Was ist Pascal?



Rolf-Dieter Klein

Dies ist kein Lehrbuch, sondern eine praxisnahe Arbeitsanleitung von Anfang an mit Pascal zu programmieren. Gerade eine schrittweise Einführung beantwortet am besten und schnellsten die Frage: Was ist Pascal?
Wer noch nie programmiert hat, wird vom Autor unmittelbar mit Pascal bekannt gemacht. Wer schon mit Basic gearbeitet hat, wird mühelos auf Pascal umgeschult. Wer schon Pascal kann, lernt mit Mikrorechner-Dialekten umzugehen.
Von anderen Pascal-Büchern unterscheidet sich dieses dadurch, daß es auf die Erfordernisse der Mikrocomputer und Heimcomputer besonders eingeht.

ISBN 3-7723-7001-2

Franzis'